Self-optimization of Infrastructure and Platform Resources in Cloud Computing

Bo Zhang

Self-optimization of Infrastructure and Platform Resources in Cloud Computing

THÈSE

soutenue le 12 Décembre 2016

pour l'obtention du

DOCTORAT Informatique

École Doctorale Sciences Pour l'Ingénieur (Lille)

Présentée par

Bo ZHANG

devant le jury composé de :

M.	Gaël THOMAS	Rapporteur
M^{me}	Patricia STOLF	Rapporteuse
M.	Noël DE PALMA	Examinateur
M.	Fabrice HUET	Examinateur
M.	Jean-Christophe ROUTIER	Examinateur
M.	Lionel SEINTURIER	Directeur de Thèse
M.	Romain ROUVOY	Directeur de Thèse

"Les pianos, c'est comme les chèques:
ça ne fait plaisir qu'à ceux qui les touchent."
—— ERIK SATIE

Abstract

Self-optimization of Infrastructure and Platform Resources in Cloud Computing

Bo Zhang

Elasticity is considered as an important solution to handle the performance issues in scalable distributed system. Particularly in Cloud computing, it has been regarded as an important property. Meanwhile, according to the rise of Big Data, many Cloud providers as well as open-source project (*i.e.*, OpenStack) begin to focus on Hadoop. They have proposed many approaches to simplify the deployment and management of Hadoop cluster for benefiting from Cloud elasticity. However, most of the researches of elasticity only concerned the provisioning and de-provisioning resources in automatic ways, but always ignore the resource utilization of provisioned resources. This might lead to resource leaks (*i.e.*, the resources are occupied but not used for processing) while provisioning redundant resources, thereby causing unnecessary expenditure. To avoid the resource leaks and redundant resources, my research therefore focuses on how to maximize resource utilization by self resource management.

In this thesis, relevant to diverse problems of resource usage and allocation in different layers, I propose two resource management approaches corresponding to infrastructure and platform, respectively.

To overcome infrastructure limitation caused by OpenStack, CloudGC — *i.e.*, a new garbage collecting system — is proposed as middleware service which aims to free occupied resources by recycling idle VMs. Moreover, cooperating with its *Recover* function, the recycled VMs can also be resumed whenever needed, thus CloudGC is able to make Cloud infrastructure support more requirements than before by successfully eliminating resource leaks in Cloud.

On platform-layer, a self-balancing approach is introduced to adjust Hadoop configuration at runtime, thereby avoiding memory loss and dynamically optimizing Hadoop performance. Finally, this thesis concerns rapid deployment of service which is also an issue of elasticity. A new tool, named *hadoop-benchmark*, applies

docker to accelerate the installation of Hadoop cluster and provides a set of *docker* images which contain several well-known Hadoop benchmarks.

The assessments show that these approaches and tool can well achieve resource management and self-optimization in various layers, and then facilitate the elasticity of infrastructure and platform in scalable platform, such as Cloud computing.

Acknowledgements

This thesis is partially supported by the Datalyse project `www.datalyse.fr`. Experiments presented in this thesis were carried out using the Grid'5000 testbed and private infrastructure of **Spirals** group, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

An immense thank you to Romain Rouvoy and Lionel Seinturier for agreeing to be my thesis advisors as well as shepherding and encouraging me through my PhD process. I truly appreciate their patience and help during these years. I would also like to thank Filip Krikava for his patience and guides at the beginning of my PhD thesis. Finally, I really thank Maria Gomez Lacruz — *i.e.*, the best officemate in the world — and her zumba.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, software elasticity becomes a popular solution in many domains. To satisfy SLA (*Service-Level Agreement*) or QoS (*Quality of Service*) constraints, the service provider requires the ability to dynamically scale the system to ensure reasonable performance. In Cloud computing, elasticity is usually confused with scalability but, compared to scalability, elasticity is much more intelligent as it also covers spontaneity, effectiveness, and timeliness issues. These features allow the elasticity to guarantee the system performance, while not causing excessive even infinite increase of the induced costs (monetary or physical).

However, even though the elasticity has many advantages and has attracted lots of attention from research and industrial communities, it cannot solve every scalability problems in software systems—*i.e.*, elasticity only focuses on the relation between system performance and resource consumption, but it does not consider the *Resources Usage Efficiency* (RUE). While elasticity might result in the waste of physical resources in order to meet SLA or QoS constraints, I believe that elasticity would clearly benefit from the integration of clever resource management heuristics in order to automatically optimize the RUE prior to the integration of any further resources. This thesis therefore explores such resource management mechanisms and heuristics that can be applied to the *Infrastructure-as-a-Service* (IaaS) and *Platform-as-a-Service* (PaaS) layers of a Cloud computing infrastructure. By focusing on these two layers, I intend to deliver reusable solutions that can be exploited by a wide diversity of applications made available as *Software-as-a-Service* (SaaS).

The contributions reported in this thesis have been conducted in the context

of the DATALYSE collaborative project.[1] DATALYSE is a French research project which addresses, but is not limited to, the *Big Data* domain. DATALYSE is interested by almost all works related to *Big Data*, from application development to infrastructure maintenance, from industrial requirements to academic researches. In this domain, there are several well-known data processing platform, such as HADOOP, STORM, SPARK. HADOOP, as the best-known platform, provides a complete ecosystem from data storage to a set of high-level applications. It is regarded as *de facto* standard in Big Data domain. STORM and SPARK focus on stream computing and in-memory computing, respectively. Compared to HADOOP, they have different designs and performances depending on case studies. DATALYSE rather focuses on HADOOP within the PaaS layer. Additionally, with the rise of Cloud computing, big data platform are more and more deployed as virtual appliances on top of Cloud platforms, like OPENSTACK. In this configuration, in order to fully benefit from physical resources, the optimizations can not only be considered within the PaaS layer, but it also requires to consider the implications on the IaaS layer.

In this thesis, I therefore focus on the resource management issues in both layers: PaaS and IaaS. More specifically, I decided to investigate resource management optimizations that can be applied in HADOOP and OPENSTACK. HADOOP is a famous distributed data processing environment in Big Data domain. It is an open-source project, promoted by the Apache consortium, which allows end-users to process large data sets across a cluster of compute nodes. HADOOP contains several modules providing a wide range of services from data storage to parallel computation. Meanwhile, *Infrastructure-as-a-Service* (IaaS) appears as an appropriate infrastructure provider for an HADOOP cluster, especially when an HADOOP cluster requires to scale on-demand. Cloud computing aims to help users flexibly obtain virtual resource upon their requirements. However, Cloud providers may tend to over-provision the physical resources to meet the expected QoS and SLA guarantees. Due to the disregard of resource usage, the scalability (even elasticity) of Cloud computing will become inefficient while causing resource waste and expenditure overruns.

[1] `http://www.datalyse.fr`

1.1 Big Picture

The big picture of this thesis is depicted in Figure 1.1. One can observe that this thesis focuses on a multi-layer processing system. Except the high-level applications, the other two layers compose our target system. In this case, the system performance is obviously affected by either infrastructure layer, platform layer, or even both layers.

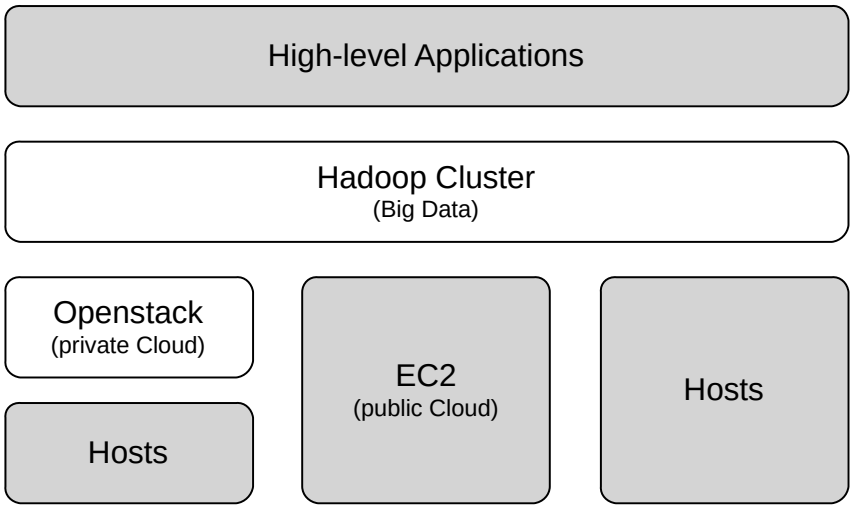


Figure 1.1: The big picture of research environment.

In this system, the infrastructure can be supported by diverse resources, such as virtual resources provided by Cloud computing or physical resources of hosts. According to the variety of resource requirements from high-level applications, Cloud computing is considered as an economical and practical choice. At the moment, there are many Cloud providers providing various possibilities for us to construct the infrastructure. To simplify the construction of our research testbed, easily reproduce the problems, and ensure the generality of our research solutions, we select the private Cloud supported by OpenStack as the achievement of the system infrastructure.

Within the platform layer, we focus on the Big Data domain. Because of the rapid growth of diverse business and the developments of terminal sensors, more and more data have to be quickly gathered and need to be processed as fast as possible. This raises many arduous challenges on relative domains of Big Data, such as data storage, analysis, processing etc. Furthermore, Hadoop—*i.e.*, a

de facto standard processing solution in Big Data domain—has attracted lots of attentions from several Cloud providers, such as AWS. Even OpenStack begins to support the Hadoop deployment and management by its sub project Sahara. We therefore choose Hadoop as the service platform to keep the pace of research and industrial communities.

Based on the big picture, we organize the contributions of this thesis along these two layers: infrastructure and platform.

1.2 Infrastructure - Cloud computing

Cloud Computing is a network-based computing that provides shared resources on demand. The main purpose of Cloud computing is to migrate the job processing and data storage from local machine to remote servers. The implementation of Cloud computing is a framework that (1) enables on-demand access to a shared pool of configurable computing resources (*e.g.*, CPU, RAM, storage, and network), (2) supports multi-tenant with high security, high availability as well as high reliability, and (3) efforts to minimize the management of virtualization technologies to leverage the provisioning and release of VMs.

Cloud providers advocate "*everything as a service*" and they structure their services in three standard models according to different layers : Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

- **Software as a Service (SaaS)** : This model aims to provide the applications installed on Cloud for the end-users. SaaS is similar to the typical service-oriented architecture. The users can access the application via website or program interface, and do not need to take care of the maintenance of applications and infrastructure. But the service providers is released from the infrastructure and platform matters —*i.e.*, the Cloud providers are responsible to maintain infrastructure and platform—and can focus on the improvement of QoS. Furthermore, the Cloud infrastructure and platform is more scalable and adjustable than the hardware. This feature is attractive to the service providers that needs to frequently scale up and down the system to satisfy fluctuating workloads.

- **Platform as a Service (PaaS)** : PaaS allows platform providers to deliver a

computing platform to application developers. The platform consists typically of operating system, specific execution environment, database, user-interfaces, etc. The application developers can develop, test, and execute their software systems on the platform without considering the underlying hardware. Some PaaS systems (*e.g.*, Microsoft Azure, Google App Engine) can automatically scale underlying computers and storage resources according to the application demand. The administrators of PaaS systems also do not need to think about the matters of infrastructure, but can control the deployed applications and the application-hosting environment.

- **Infrastructure as a Service (IaaS)** : IaaS model permits users to control diverse resources, such as network, operating system, database, VM type, VM number, etc. The users must customize and maintain the VMs to support their platform as well as software like administrators of actual infrastructure. However, IaaS model prevents Cloud users from dealing with numerous details of infrastructure like machine maintenance, location, data partitioning, security etc. Benefiting from "pay as you go" billing model, Cloud-users can also more easily tune the infrastructure than before.

Cloud computing can help the end-users to avoid upfront infrastructure costs, and to focus on their jobs instead of on infrastructure. Furthermore, Cloud computing also permits end-users to rapidly and easily adjust their "infrastructure" to meet varying workloads. And compared to the real materials, the maintenance of the "infrastructure" (*i.e.*, VMs) is much less and easier. Besides these significant advantages, thanks to wide-bandwidth network, low-cost VMs, huge storage volumes, concise user interfaces as well as the rapid increase of computing needs, Cloud computing has become a reasonable choice for many service providers.

As Cloud computing become more and more popular, the relative researches of Cloud computing also attracts lots of attention from research and industrial communities, such as elasticity. Even though many Cloud providers want their users to consider the Cloud as an infinite resource pool, the limitation of infrastructure still exists, particularly in the case of private Cloud (*i.e.*, OpenStack Cloud). The primary idea of researchers and developers is to improve the elas-

ticity of Cloud computing for dynamically provisioning resources to meet users' requirements. However, due to stiff resource allocation mechanism and infrastructure limitations of OpenStack, the elasticity in Cloud infrastructure may cause redundancy of physical resources (over-provisioning), increase the operating cost of Cloud as well as other expenses, such as energy consumption and carbon emissions. To avoid this problem, I propose an approach in Chapter 4, named CloudGC, to recycle the idle resources in Cloud to serve more users when the total requirement reaches the limitation of Cloud infrastructure.

1.3 Service Platform - Hadoop

Hadoop is an open-source project which aims at providing reliable, scalable, and distributed computing to deal with large datasets. This project is designed in a master-slave architecture, which can be easily extended from a single machine to thousands of compute nodes. Hadoop includes 4 core modules:

- **Hadoop Common** is the base library supporting all the other modules.

- **Hadoop Distributed File System (HDFS)** is a distributed file system providing storage service for Hadoop applications across clusters. HDFS is also designed as a master-slave architecture. It consists of 1 NameNode and a set of per-node components DataNode. NameNode manages the file-system metadata, which keeps track of the location of actual data in HDFS. DataNode is responsible to store the actual data in the disk of compute nodes. This architecture helps users to quickly locate the desired data without browsing the entire file-system, resulting in high-throughput access to application data.

- **Hadoop YARN** is a scheduling framework included in Hadoop since an overhaul of Hadoop architecture in 2012. YARN focuses on job scheduling and cluster resource management. The MapReduce or another high-level framework therefore can build on YARN to ensure the job processing. However, due to the static configuration and the defects of architecture, YARN may degrade the Hadoop performance instead of optimizing it. This problem and my proposition will be further explained in Chapter 5.1.

- **HADOOP MapReduce** is a famous parallel processing framework. This module has been updated to a YARN-based system in Hadoop2 since 2012. But its processing paradigm remains the same as before—*i.e.*, MapReduce processing paradigm contains Map and Reduce phases, and the intermediate results is still organized by a shuffle phase. As a famous and popular processing framework, I used this module as the application (service) of HADOOP cluster in this thesis.

Next to the core modules, HADOOP has a complete ecosystem. The other Hadoop-related projects include Ambari, Avro, Cassandra, Chukwa, HBase, Hive, Mahout, Pig, Spark, Tez, ZooKeeper, etc. The ecosystem of HADOOP is shown in Figure 1.2.

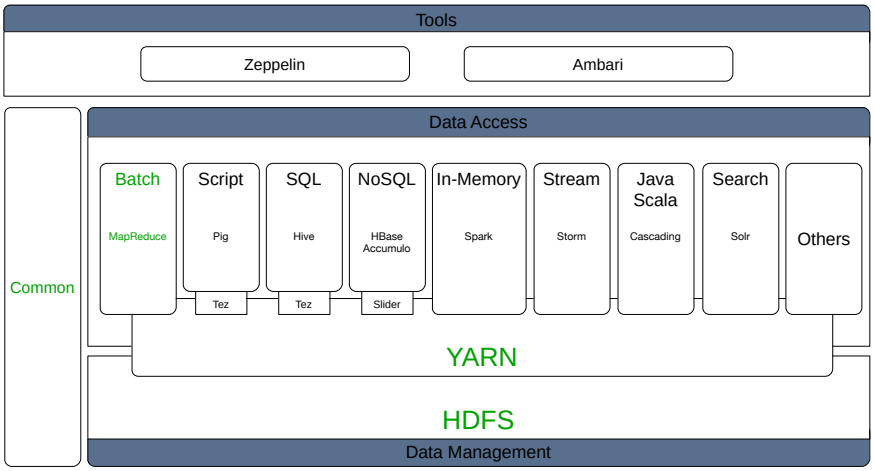


Figure 1.2: Ecosystem of Hadoop.

Because of the scalability of HADOOP cluster, many service provider begin to transfer HADOOP service to Cloud for benefiting from IaaS, such as HADOOP on Google Cloud Platform [2]. However, due to the static configuration and the defect of architecture, HADOOP cluster may waste resource when processing concurrent workloads, resulting in loss of performance. To solve this problem, we focus on the resource management of HADOOP cluster and propose an approach to dynamically adjust HADOOP configuration at runtime in Section 5.1.

[2]https://cloud.google.com/hadoop

1.4 Resource Optimization

In Cloud computing, elasticity is considered as an ability which permits the system to automatically adjust to the workloads. It is an important criteria to distinguish Cloud computing from the other computing paradigms, such as grid computing. Elasticity aims to quantify the resource allocation for avoiding the resource over- and under-provisioning in an autonomous manner.

- **Over-provisioning** means excessive resources allocated compared to the requirements. This can help the system to guarantee the performance, but also induces an high expenditure which may degrade the user experience, resulting in low QoS. Meanwhile, for the service providers, over-provisioning leads to the cost increase (*e.g.*, physical machines, energy consumption)—*i.e.*, to satisfy their users, they must pay a higher cost than the expected one.

- **Under-provisioning** is contrary to over-provisioning. It can save the cost to both end-users and service providers. However, under-provisioning is the typical cause resulting in the loss of performance and has grabbed lots of attention.

In details, the elasticity of Cloud computing has several properties : *spontaneity*, *effectiveness*, *timeliness*, and *scalability*.

Spontaneity decides autonomously when the system needs to be scaled up or down. The moment determined by *spontaneity* must conform 2 conditions: (1) the current system performance cannot satisfy user requirements, (2) the performance degradation is certainly caused by the lack of resource. To be considered as "elastic", a system must satisfy the premise that it can autonomously adapt to the workloads only at the necessary moment. This premise is an important problem to elasticity researches, but also the most forgotten part. My thesis aims at focusing on this problem and complementing the current elasticity researches.

However, only supporting *spontaneity* is far from becoming an elastic system. The main purpose of elasticity is to avoid either over- or under-provisioning of resources. In this case, the precise amount of resources to be provisioned or deprovisioned becomes a key problem in elasticity. *Effectiveness* therefore is one of the core properties in elasticity. Furthermore, *timeliness* is also important.

Both end-users and service providers all expect the elasticity can help systems to optimize the performance. But a long period of provisioning or de-provisioning will certainly degrade the impact of elasticity. Finally, *scalability* is a classical topic in many domains.

These properties are corresponding to several classical problems in elasticity.

- **Resource Allocation Delay** impacts directly the effect of elasticity. In Cloud, the resource is provided in the form of VM (*virtual machines*). Even though end-users can require VM at any time, the acquirement of VM is not instantaneous. The VM is not available until its operating system becomes ready. The deployment time of VM depends on many factors, such as its flavor (VM size), image size, network connection, infrastructure congestion, number of provisioned VMs, etc.

- **Precise Scalability** requires deep analysis of the applications deployed on the Cloud. For various applications, the trade-off between system performance and resource consumption will be widely different. Moreover, this trade-off is also difficult to be quantified in real world.

- **Application Monitoring** is also a concern of elasticity. Due to the elasticity, the resources allocated to the applications become volatile—*i.e.*, in Cloud, the VMs occupied by applications can be dynamically added or removed. The traditional monitoring tools based on fixed clusters are no longer suitable to elastic systems, such as Ganglia or Nagios. Furthermore, the variable cluster size (*i.e.*, number of VMs) of applications proposes new challenges to the aggregation of the metrics (*e.g.*, the mean of CPU utilization).

- **Multi-Level Control** reveals a new challenge of elasticity in Cloud computing. Besides the impact on application-level, the provisioning and de-provisioning behaviors also affect the performance of Cloud computing which will be exposed in Chapter 4. Any control in low-layer (*e.g.*, infrastructure) may cause unpredictable impact upon high-layer, and vice versa.

The researches about elasticity in Cloud computing realize wide coverage across layers (from infrastructure to application) and across domains (from monitoring

to scalability). However, as I enumerated above, the core problems of elasticity do not concern the resource utilization—*i.e.*, most of these researches only focus on the performance optimization, but ignore the usage improvement of provisioned resources. This is prone to cause over-provisioning and degrade the benefits of elasticity. As I explained before, this thesis therefore focuses on the resource optimization of provisioned resources to achieve the optimal performance that the current infrastructure can. This can also be regarded as a part of spontaneity researches in elasticity, or an complementation of elasticity.

The resource optimization aims at improving the system performance by eliminating limitations (*e.g.*, misconfiguration) based on provisioned resources. Even though the QoS cannot satisfy the user demand, the resource optimization can reveal the necessity of additional resources to guarantee the performance, and improve the effect of elasticity. In this case, the resource optimization is therefore an important complement of elasticity, especially to improve the spontaneity of elasticity.

1.5 Thesis Contribution

In this thesis, I first focus on the resource management to improve the resource usage at runtime. The service performance and infrastructure efficiency will therefore be improved. And then, based on the definition of elasticity, my research concerns the rapid deployment of service, which is an important part of service elasticity. The contribution of this thesis can be divided into 3 parts —*i.e.*, 1 focusing on OpenStack (infrastructure) and the other 2 concerning Hadoop (platform):

(1) an OpenStack garbage collecting system, named CloudGC, is realized to recycle the idle VMs in Cloud. This new middleware service achieved by CloudGC lets the Cloud providers to serve more end-users within the same Cloud infrastructure.

(2) a self-balancing approach is proposed to improve job parallelism and throughput at runtime to elevate the memory utilization of Hadoop cluster, resulting in the performance optimization.

(3) a rapid deployment tool to accelerate the installation and expansion of provisioned HADOOP clusters. This tool also aims at facilitating the related researches of elasticity of HADOOP cluster.

CLOUDGC is an OPENSTACK garbage collecting system. As previously introduced, OPENSTACK is a large open-source project to build Cloud platforms. The Cloud supported by OPENSTACK contains lots of tools which provide a wide range of services, such as resource allocation, resource monitoring, VM scheduling etc. In the course of processing multi-tenant requires, the resource allocation mechanism of OPENSTACK expose a serious problem that the resource allocation ignores the actual resource utilization—*i.e.*, when the resource occupation reaches the infrastructure limitation (or quotas limitation), any new requirement cannot get the resource even though the Cloud is actually idle. To address this problem, I propose a new system, named CLOUDGC. It aims at recycling the idle VMs to eliminate the idle resource occupation, thereby allocating resources to those who really need them. The Cloud can therefore satisfy more requirements than before, without scaling up the infrastructure. CLOUDGC also contains several strategies to handle *explicit* and *implicit* idle VMs together. From the evaluations, one can recognize that CLOUDGC can help Cloud surpass its infrastructure limitation when idle VMs appear. The proof of motivation and the detail of CLOUDGCwill be illustrated in Chapter 4.

Chapter 5 introduces a new self-balancing algorithm which aims to improve memory utilization of HADOOP cluster. HADOOP is a famous distributed data-processing environment in Big Data domain. It has a complete ecosystem and a large set of high-level applications concerning many research and industrial fields, such as data mining, machine learning, distributed storage etc. However, after lots of experiments, we observed that (1) the static HADOOP configuration cannot guarantee the optimal system performance for different workloads, and (2) the HADOOP performance will expose a non-monotone behavior according to the change of configurations. Based on the understanding of HADOOP architecture and the analysis of memory consumption, I propose a self-balancing algorithm to adjust HADOOP configurations at runtime, thereby improving memory utilization of HADOOP cluster when processing concurrent workloads. The approach

containing this algorithm is implemented in a feedback control loop. The evaluations reveals that this approach can significantly optimize the memory usage when HADOOP cluster handles concurrent workloads, thereby improving system performance. The details will be further explained in Chapter 5.

Finally, a new tool, named *hadoop-benchmark*, is proposed to accelerate the deployment of HADOOP cluster. This tool benefits from *Docker container technologies*, which aims to achieve lightweight application deployments. Using the *Docker* image service, *Docker container technologies* can quickly provision an operating environment for applications, where all the dependencies execute. In my case, I package a complete HADOOP cluster into several images (*e.g.*, images for control-node, for compute-node, and for several well-known HADOOP benchmarks) I evaluate this tool with virtualbox in a single node, Microsoft Azure, and Grid5000. THe evaluation show that *Hadoop-benchmark* can simplify and accelerate the deployment of complete HADOOP cluster on different infrastructures. This is important to prove that *Hadoop-benchmark* can facilitate the researches of elasticity in various scalable environment, such as Cloud computing.

1.6 Thesis Organization

The rest of this thesis is organized as follow. The part 2 (State of the Art) consists of two chapters. Chapter 2 exposes the related work of infrastructure-level optimizations in Cloud computing. And the state of the art for platform-level optimizations, related to Hadoop, is introduced in Chapter 3. Then, in the next part (Contribution), Chapter 4 illustrates a resource loss problem in OPENSTACK Cloud and introduces a new middleware system, named CLOUDGC (cf. This research has been accepted as full paper by IC2E 2017 [3]), to improve the resource utilization. Next to the resource optimization of infrastructure, Chapter 5 presents 2 Hadoop-related works: (1) a self-balancing approach [63, 64] of HADOOP cluster, based on YARN, to improve the memory utilization at runtime, thereby optimizing the QoS of HADOOP cluster when processing concurrent workloads; (2) *Hadoop-benchmark*, a rapid deployment tool of HADOOP cluster with

[3] http://conferences.computer.org/IC2E/2017/

a set of well-known HADOOP benchmarks, is proposed in this chapter. Finally, I conclude in Chapter 6 and finish this thesis with my perspectives in Chapter 7.

Part : State of the Art

Chapter 2

Infrastructure-level Optimizations

As a kind of network-based computing, Cloud computing can be regarded as a general term for the on-demand delivery of remote computing resources. It enables Cloud users to consume computing resources as a utility rather than having to build and maintain local infrastructure. Within this shared resource pool, Cloud users can flexibly allocate compute resources from the Cloud and freely release them on demand. Compared to local infrastructures, Cloud computing can significantly reduce the expenditure and liberate resource users from the tedious maintenance. This makes Cloud computing a preferable choice for those needing the resources only for a short term or having no ability to build local infrastructure. The beneficiaries are distributed along a wide range, from large companies to new entrepreneurs. In this case, Cloud computing becomes an attractive topic in industrial and research communities. Many companies and research institutes have provided various implementations, such as public cloud (*e.g.*, Amazon EC2) and private cloud platform (*e.g.*, OpenStack).

OpenStack is an open source project, which is released under the terms of the Apache License[1]. It began in 2010 as a joint project of Rackspace Hosting[2] and NASA[3]. OpenStack does not only help users to easily and quickly create private cloud, but it also permits users to customize the cloud on demand. This also allows many researchers to go deeper into the architecture of the Cloud to identify problems, explore new solutions, integrate and experiment them on private Cloud testbed. As an open source project, OpenStack is therefore important for the

[1]http://www.apache.org/licenses/LICENSE-2.0
[2]http://www.rackspace.co.uk
[3]https://www.nasa.gov

development of Cloud computing.

The researches of Cloud computing focus on several characteristics:

- **Cost** is an important factor for stakeholders, both Cloud providers and users. Many researchers try to help users minimizing the cost of their consumptions, while maximizing the performance in a complex computing environment, such as hybrid or heterogeneous Cloud. They provides diverse methods to compare the possible performance and unit expenditure, with various combinations to determine the optimal strategies of resource allocation. In contrary, Cloud providers also want to serve more users with fewer resources, which take lower costs (*e.g.*, monetary expenditure, energy consumption).

- **Scalability** or **Elasticity** is a popular topic in Cloud computing, which indicates the ability of Cloud computing to autonomously and dynamically manage resources. These researches involve lots of areas, such as self-adaptation, cybernetic, etc. This characteristic is an important indicator to differentiate Cloud computing from the other cluster or grid computing.

- **Security** does not only protect the user privacies, but also concerns the normal operation and maintenance of Cloud computing. Beyond the algorithm development and architecture optimization, the multi-tenancy and resource sharing also highly affects these researches.

- **Reliability** improves Cloud computing by strengthening its ability to resist risks. This is important to Cloud computing, particularly when there are congestion or over-loads which easily result in the failure of user requests.

- **Agility** allows Cloud users can quickly provision or de-provision resources, which obviously affects the user experience.

- and etc.

2.1 OpenStack

The Cloud can be regarded as a resource pool which provides computing services for end-users in a remote environment. The Cloud computing supports

different items "as a service" (*e.g.*, software, platform, infrastructure)—*i.e.*, SaaS (*Software-as-a-Service*), PaaS (*Platform-as-a-Service*), and IaaS (*Infrastructure-as-a-Service*) are the three typical services of Cloud computing. OpenStack belongs to the latter category and is considered as a reference implementation of a IaaS. OpenStack is an open-source Cloud operating system that contains a set of software tools for Cloud computing services of public and private Clouds. It aims to help Cloud providers to dynamically manage the resources and to simplify the procedure of resource allocation for end-users. OpenStack can manage various computing resources (*e.g.*, compute, storage, networking) and can regroup them by the form of VM. It allows end-users to customize the VM with their favorite "flavor"—*i.e.*, a template of VM which has defined CPU, RAM, and disk informations—and *Operating System* (OS).

OpenStack contains several different moving components. Because of its open-source nature, the Cloud providers can enable, disable, modify, and even add new components into the architecture to meet their needs. However, there are 9 core components identified by the OpenStack developers. The architecture of OpenStack is shown in Figure 2.1.

I) ***Nova*** is the principal component in OpenStack. It can be considered as Cloud computing fabric controller, which supports variety of virtualization technologies (*e.g.*, KVM), as well as bare metal and *High-Performance Computing* (HPC) configurations. The IaaS is mainly achieved by *Nova*. *Nova* is also designed as a computing engine, which serves to provision and to manage large sets of VMs on demand.

II) ***Glance*** is responsible for image services in OpenStack. For virtualization technologies, images refer to virtual copies of physical machines, which can be regarded as a template when deploying new VM. The services supported by *Glance* include discovering, registering, and retrieving images in OpenStack. Furthermore, end-users can also use *Glance* to backup a live VM.

III) ***Swift*** is a highly available, distributed, and consistent storage system for objects and files. It is designed to store and retrieve lots of unstructured

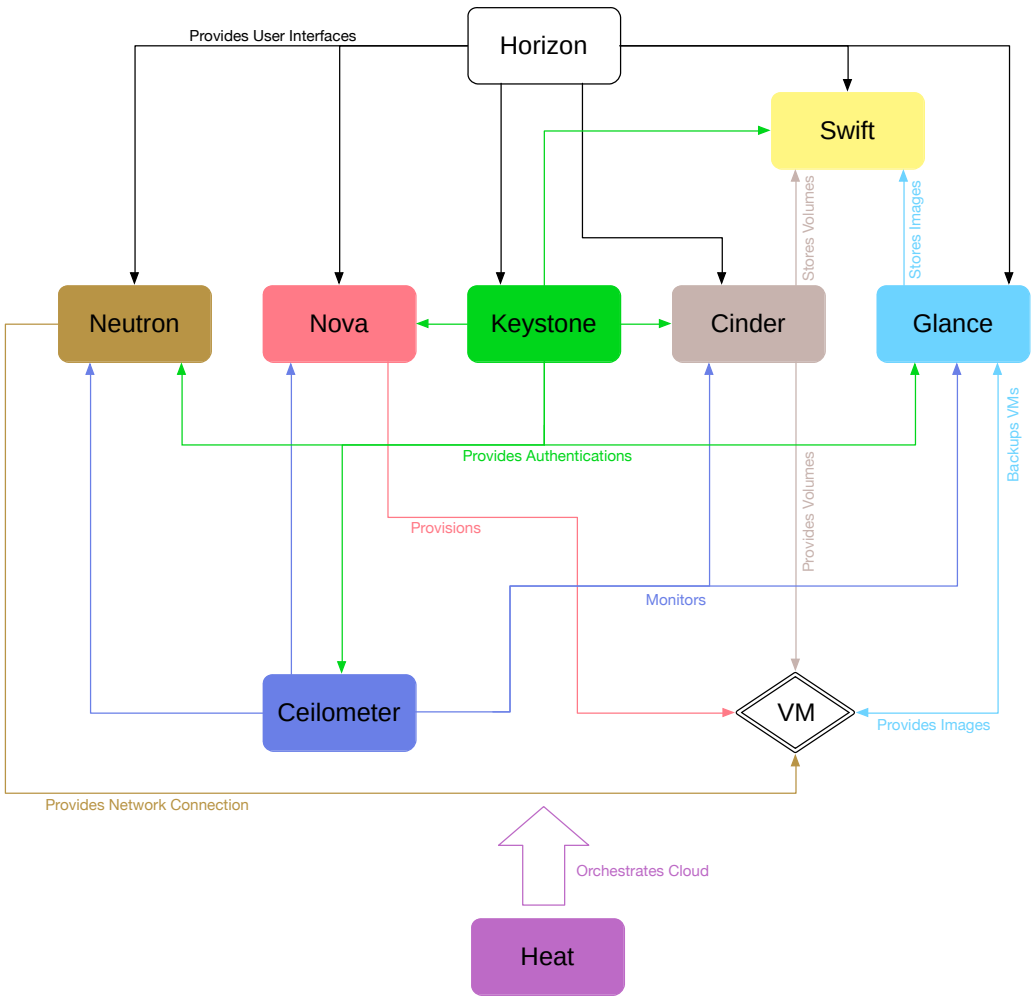


Figure 2.1: Architecture of OpenStack

data with a sample API. *Swift* primarily serves to store the volumes created by *Cinder* or the images registered by *Glance*. *Swift* let the system —*i.e.*, OpenStack in this case, rather than users, decide where to store the data. End-users therefore also needn't to worry about how best to back up the data in case of the failure of compute nodes or network connection.

IV) ***Ceilometer*** is a monitor component in OpenStack. It regularly reports various metrics of the other components and the resource usage in OpenStack. Moreover, *Ceilometer* can also expose the resource utilization (*e.g.*, CPU utilization) of each VM in Cloud. The CPU utilization of VM, collected by *Ceilometer*, is an important metric for CloudGC to detect the idle VM.

Besides these 4 core components , OpenStack still contains ***Keystone*** which

provides authentication service to each other component and data center of OpenStack, ***Neutron*** ensuring the network connection among all the other components and VMs, ***Cinder*** which arranges volume services, ***Horizon*** dashboard of OpenStack, and ***Heat*** that orchestrates OpenStack.

Furthermore, OpenStack also contains lots of additional components, like **Mistral** which is used to manage workflows, **Trove** being a database-as-a-service engine, **Ironic** supporting bare metal provisioning service, **Manila** which is an OpenStack shared file system, **Designate** managing DNS for multi-tenant, **Searchlight** providing advanced and consistent search capabilities across various OpenStack cloud services, **Zaqar** supporting messaging service for multi-tenant, and **Barbican** for secure storage etc. In particular, **Sahara** is a component for managing Hadoop cluster to achieve Elastic MapReduce service. This component shows the ambition of OpenStack communities and their willing to build the connection between Cloud computing and Hadoop-based services. In this case, **Sahara** also exposes the increasing importance of Hadoop system in Big Data domain.

2.2 Infrastructure-level Optimizations

To the best of our knowledge, no approaches have been proposed so far on developing an optimization inspired from garbage collectors and applied to the Cloud, which is the infrastructure-level optimization we target. The most similar approach is Heroku[4], which operates at the *Platform-as-a-Service* (PaaS) level. In Heroku, Cloud services deployed with the free offer automatically fall asleep after 30 minutes of inactivity, and are resumed on-demand, inducing a similar delay to CloudGC for accessing the service again. At the IaaS level, several elasticity solutions have been developed by public providers and academia. In the remainder of this section, we will therefore present and discuss some of the closest related works for IaaS that are relevant to our approach.

[4] `https://www.heroku.com`

2.2.1 Resource Management

Recycling resources is the process of collecting, processing, and reusing *Virtual Machines* (VMs). Current state-of-art Cloud infrastructures, such as OpenStack [46], do not support recycling inactive or ghost VMs to maximize the amount of allocable resources for a given user. As a result, inactive nodes tend to stay around longer and unnecessarily consume more resources than when an efficient reclamation process is in place.

There are many proposed systems using resource management for various computing areas [12, 24, 49, 51], but none of these systems focuses on the problem of recycling VMs in addition to maximizing resource utilization. Then, these systems are unable to identify inactive nodes and initiate the process to recycle the nodes' resources.

Suraj *et al.* [49] introduce a PSO-based (*Particle Swarm Optimization*) scheduler to assign applications to Cloud computing. This scheduler does not only consider the cost of VMs allocated in a cloud, but it also takes data transmission into account, which concerns the network exchanges among VMs and external users. Thanks to this scheduler based on a PSO algorithm, users can deploy their applications in the cloud computing with the lowest expenditure, the most efficient energy consumption, or both of them. From the comparison between this scheduler and other approaches based on *Best Resource Selection* (BRS) algorithms, the total cost of application in a cloud has been saved by at least 1/3. However, rather than continuous monitoring and resource utilization optimization of a cloud, this scheduler only affects the distribution or the resource allocation for the user applications.

Rajkumar *et al.* [12] do not only present vision, challenges, and architectural elements for energy-efficient management of data centers, but also propose resource allocation policies, scheduling algorithms as well as a novel software technology to achieve Green Computing in Cloud computing environments. Dang *et al.* [51] also focus on the resource allocation strategy of data centers deployed on Cloud computing. This research aims at minimizing the energy consumption of data centers on Cloud computing while meeting the demands for highly-responsive computing and massive storage. Compared to elevating resource utilization and cleaning un-

necessary resource occupancy, the authors are more interested on how to make sure the data center can allocate the suitable VMs which have enough capacity to satisfy user requirements while consuming the least energy.

Compared to the three above resource management researches on Cloud computing, Sijin *et al.* [24] also concentrate on the resource utilization of VMs. However, this research is only attracted by the limitations of resource utilization, which are caused by misconfiguration of VMs and physical hosts in Cloud infrastructure. The proposed solution cannot address the resource leaks caused by idle VMs. Therefore, this research is able to improve the resource utilization in VMs but cannot elevate the resource utilization of Cloud infrastructure.

Furthermore, existing energy-efficient resource allocation solutions proposed for various computing systems [5, 13, 62] also cannot be using for recycling VMs. This is because they only focus on minimizing the energy consumption or their costs, and do not consider dynamic service requirements of consumers that can be changed on demand in Cloud computing environments.

2.2.2 Cloud Elasticity

As one of the most important features of Cloud computing, elasticity has grabbed lots of attention from industrial and academic communities. Many researchers and Cloud providers have proposed various approaches and services to realize Cloud elasticity at runtime to improve the QoS of Cloud computing.

These propositions can be separated into two categories: **Horizontal elasticity** and **Vertical elasticity**.

- Horizontal elasticity is the typical elasticity which focus on tuning the amount of VM instances to control the performance of applications deployed in the Cloud. It is similar to the scalability but in a more intelligent manner, as it does not only contain the scalability, but also concerns the spontaneity and timeliness of scaling actions.

- Vertical elasticity can be regarded as a more fine-grained elasticity. These researches concentrate on the adjustments inside VM instances. According to the dynamics of workloads, the provisioned resources will be extracted (or added)

from (or to) the existing VM instances, such as modifying the type of VM instances (*e.g.*, transforming *small* instances to *tiny* ones).

2.2.2.1 Horizontal elasticity

Horizontal elasticity consists in adding/removing instances from the user virtual environment to ensure the application performance while controlling the user cost.

Beernaert *et al.* [6] proposed an adaptation and monitoring component for IaaS atop of Cloud infrastructures: Elastack. Elastack is an example of horizontal elasticity solution. It permits to add or remove VM instances according to the dynamics of workloads. It continuously monitors all the VM instances on compute nodes and then, based on the total CPU usage, the VM instances are added or removed from the Cloud infrastructure to minimize the user cost while the provisioned VM instances are able to support the QoS of user applications. One can find that Elastack does not take into consideration the problems occurring when user requirements reach infrastructure limitations: what will happen and how to resolve or avoid this situation?

Kaleidoscope [11] is a similar work, but instead of launching a new instance from scratch to satisfy a workload demand, the new VM instances are cloned with a copy of complete or partial state of the existing VM instances. This can simplify the configuration of new VM instances to degrade the negative impact of misconfiguration, but the creation period will be prolonged. Furthermore, the copy period might also affects the existing VM instances, thereby decreasing the system performance of the VM instances.

Amazon WS [3] also allows users to dynamically add or remove VM instances according to the needs. In addition, it provides a load balancer to distribute load between different instances.

2.2.2.2 Vertical elasticity

Vertical elasticity consists in adding or removing resources of individual VMs supporting the applications. Several works have been done on vertical elasticity. Some works focus on CPU resizing, and others concentrate on memory resizing as well as combinations of both.

CloudScale [54] is an automatic elastic resource scaling system for multi-tenant infrastructures. It provides an online resource demand prediction and an efficient prediction error handling mechanism. It allows instances vertical scaling with *Dynamic Voltage Frequency Scaling* (DVFS) to save energy. In addition, it permits to solve the problem of conflicts when the available resources are insufficient to accommodate instances scaling up requirements on a host.

In [55], a model-based approach to vertical scaling of vCPUs is proposed. This approach allows resource allocation to the associated VMs in accordance with the application *Service Level Objectives* (SLO).

Farokhi et al. [20] designed controllers that dynamically adjust the amount of CPU and memory required to meet the performance objective of an application for a given instance. This work did not only propose CPU and memory controllers to adjust vertical elasticity for each, but also proposed fuzzy controller that acts as a coordinator between the two controllers to adjust elastic actions of one controller to the other.

However, for the current version, OpenStack does not support vertical elasticity yet and, to the best of our knowledge, there is no literature which examines whether OpenStack supports vertical scaling [56].

2.2.3 VM Consolidation

VM consolidation is an approach towards a more efficient usage of resources in order to optimize the number of servers that an organization requires. OpenStack controls instance placements on compute nodes via the component nova-scheduler. In particular, the Filter scheduler supports filtering and weighting to make informed decisions on where a new instance should be created. OpenStack can be configured to allow host-level control of VM placement. This can be done by defining an availability zone in the configuration of each host. The identifier of the availability zone can be passed to OpenStack as a parameter in a compute create request; the OpenStack internal scheduler will then place the resource appropriately. The VMs placement on physical servers has to take in account several considerations, such as power consumption and performance metrics.

For example, Corradi et al. [17] proposed a Cloud management platform for

OpenStack to optimize instances consolidation along three dimensions: power consumption, host resources, and networking. This work runs experiments to evaluate the runtime side-effects introduced by consolidating VMs on OpenStack. It showed that VM consolidation is convenient to reduce power consumption, but could leads to performance degradation.

OpenStack Neat [7] is a framework for dynamic consolidation of VMs in OpenStack clouds. The objective of this system is to reduce power consumption and to avoid performance degradation. The system decides whether a compute host is underloaded, so that all the VMs should be migrated from it, and it will be switched to sleep mode to save energy. If a compute host is overloaded, some VMs will be selected to be migrated and placed on other hosts in order to avoid performance degradation. Efficient algorithms decide VMs migration and placement. The system uses Wake-on-LAN [58] technology in order to reactivate hosts in sleep mode if it decides to migrate new VMs to them. The framework migrates VMs from on compute host to another but it does not not take in account the limitation of the infrastructure.

Hong et al. [29] studied also the problem of efficiently consolidating many VMs on a physical server with keeping the best trade-off between the Cloud provider profits and client quality of experience. Quality-driven heuristic algorithms are proposed to choose on which physical servers to deploy VMs based on the network latency to the client.

More recently, Oasis [67] scales the cluster size by making the idle hosts fall in sleep to save energy. During low resource utilization periods, Oasis migrates and centralizes the VMs to low-power servers. This therefore 1) guarantees the running of VMs and 2) allows the idle hosts to rest in sleep mode. Oasis is an asynchronous migration approach. It partially migrates the idle VMs to consolidation host firstly, and then making the idle host to fall asleep. When the VM requires all its resources, the left part of the VMs will be migrated to the consolidation host from the awakened home host. The main idea of Oasis is therefore to reduce the power consumption, but not to improve the resource utilization—*i.e.*, the number of VM cannot exceed the limitation of infrastructure. Thanks to Oasis, the dense server (host) consolidation can be used to save energy, thereby reducing the cost

of whole cluster.

PowerNap [41] can rapidly tune system between high-performance state and near-zero-power state in response to instantaneous workloads, particularly in the case of frequent and brief burst of activities. When a server exhausts all the pending jobs, the host will become nap state (*i.e.*, near-zero-power state). In this state, the power consumption become very low and no processing can occur. Until NIC detects the arrival of new requests, the host will be woken up and return to active state (high-performance state).

2.3 Synthesis

From the above related works, we can find that most of the recent researches of Cloud computing focus on resource allocation, elasticity or VM consolidation to optimize the system performance while minimizing the cost, such as power consumption. Even though these researches are able to guarantee the QoS of Cloud computing, their effectiveness is limited due to the disregard of resource utilization of Cloud infrastructure. This therefore might cause resource waste and result in unnecessary expenditure. To avoid the resource waste in Cloud infrastructure (*i.e.*, the idle resource occupation of idle VM instances), a new middleware service is proposed to recycle idle VM instances to re-allocate the Cloud resources for the other urgent requirements, thereby improving resource utilization. This middleware service is implemented as a dedicated component integrated in OpenStack, which will be introduced in Chapter 4.

Chapter 3

Platform-level Optimizations

Big Data is not only a term talking about the size of datasets (both structured and unstructured data), but also a broad domain consisting of diverse issues, researches, and applications in academia and industry. Due to the large volume of datasets collected by numerous sensors (*e.g.*, mobile), the processing and management of these data become more and more difficult to guarantee the performance and QoS. In this case, the integration of new techniques and technologies is required to overcome various challenges in Big Data domain, such as data analysis, data storage, distributed processing, etc. Hadoop is therefore proposed as a software solution to facilitate the development of Big Data applications. Thanks to its evolution along years and the complete ecosystem, Hadoop has become the *de facto* distributed data processing platform in Big Data domain.

According to the rise of Hadoop, industrial and research communities pay more attention to this distributed data processing environment. Many large companies begin to provide MapReduce services, such as Amazon Elastic MapReduce (Amazon EMR) using Amazon EC2 as underlying cloud infrastructure. OpenStack also propose a subproject, named *Sahara*, to provide a simple means to provision Hadoop cluster on top of OpenStack Cloud. These services and projects obviously enrich the application environment and testbed of Hadoop-related researches, that in turn lowers the *barriers to entry* of Big Data domain.

Benefiting from IaaS of Cloud computing, many researches are able to scale Hadoop cluster to keep the pace of dynamics of workloads, thus protecting the system performance. In this case, scalability and elasticity of Hadoop cluster become attractive topics in Big Data domain. Moreover, due to the complexity

of Hadoop projects and hundreds of configuration parameters, many researchers also advocate various optimization approaches in different domains, such as self-adaptation, data storage, etc. In this chapter, I enumerate several Hadoop-based approaches, which can be considered as mainstream researches.

3.1 Hadoop

Hadoop is an open-source project for distributed storage and distributed processing of large datasets. The data storage service is organized by HDFS (*Hadoop Distributed File System*), which is a distributed system across multiple machines. It adopts the strategies to (1) separately store the actual data and its corresponding metadata in different components, and to (2) replicate the actual data across multiple machines. This provides high-throughput access to the datasets and guarantees the reliability of the database. Since 2012, Hadoop has introduced a new component, named YARN, which is specifically responsible for scheduling applications. The scheduling of high-level applications in Hadoop cluster is not anymore an exclusive service packaged in MapReduce paradigm, but a Hadoop core service for all Hadoop-based applications. The two components (*i.e.*, HDFS and YARN) of Hadoop basis will be further described in next section.

3.1.1 Hadoop Basis

Hadoop Basis is composed of the two main subsystems: HDFS (*Hadoop Distributed File System*) and YARN (*Yet Another Resource Negotiator*). The high-level architecture is depicted in Figure 3.1. HDFS is an implementation of a distributed file-system that stores data across all compute nodes in Hadoop cluster. It applies the master-slave architecture to achieve the unified database management in the distributed system, while ensuring high throughput access to the stored datasets. Typically, HDFS uses 1[1] NameNode server that hosts the file-system metadata and a variable number of DataNode servers that store the actual data blocks. This architecture provides an overview of database storage for the users and obviously reduces the access time of all data sets. Benefiting from the metadata provided by NameNode, users can quickly locate and access to the target

[1]coupled with a secondary instance for high availability.

data, rather than browsing the whole database. For the Big Data requests which probably process a large set of data, HDFS can significantly accelerate the process of these jobs. YARN is a cluster-level computing resource manager, which is responsible for resource allocations and jobs orchestration. It consists of 1 per-cluster ResourceManager that acts as a global computing resource arbiter and a per-node NodeManager which corresponds to manage node-level resources.

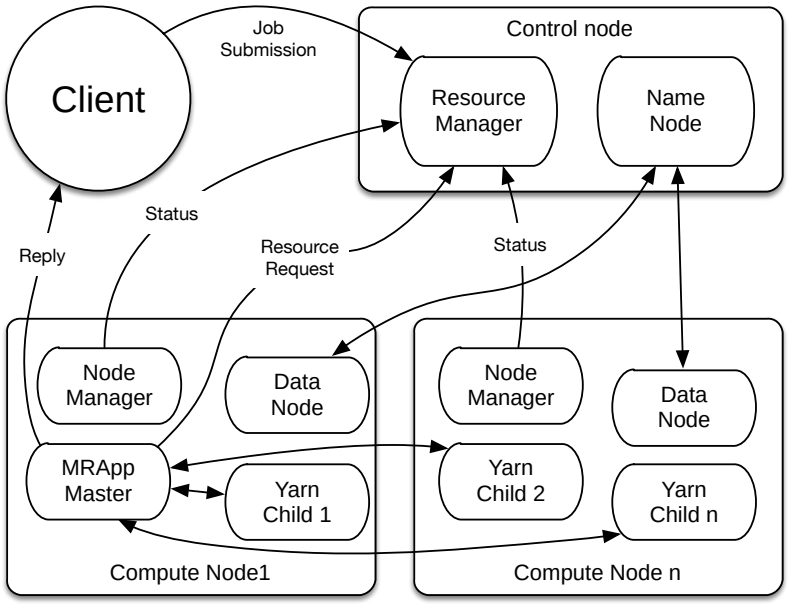


Figure 3.1: High-level Hadoop architecture.

Usual Hadoop cluster contains one `Controller` node dedicated to the ResourceManager and NodeManager and a number of `Compute` nodes for the workers running NodeManager and DataNode.

3.1.2 YARN

YARN is a cluster-level computing resource manager responsible for resource allocations and overall jobs orchestration. It provides a generic framework for developing distributed applications that goes beyond the MapReduce programming model. It consists of two main components (cf. Fig. 3.2): (1) a per-cluster ResourceManager acting as a global computing resource arbiter and (2) a per-node NodeManager responsible for managing node-level resources and reporting their usage to the ResourceManager.

The main difference from the previous Hadoop versions (up to 0.23) is that YARN separates the resource management tasks from the application management tasks and therefore allows to run heterogeneous jobs within the same cluster. The ResourceManager contains a scheduler, named capacity scheduler, that allocates resources for the running applications. However, it does not do any application monitoring or status tracking. This responsibility is left to the per-job instance of *Application Master* (AM). AM is an application-specific process that negotiates resources from the ResourceManager and collaborates with the NodeManager(s) to execute and monitor its individual tasks. The scheduling is based on the application resource requirements and it is realized using an abstract notion of *containers*. Essentially, each computing node is partitioned into a number of containers which are fixed-size resource blocks (currently only the memory is supported) running AMs and their corresponding tasks.

For example (cf. Fig. 3.2), for a MapReduce application, ResourceManager deploys two type of containers: MRAppMaster and YarnChild. When a job arrives (1), the ResourceManager negotiates the first container for executing its specific AM process—*i.e.*, MRAppMaster. The MRAppMaster is then responsible for negotiating additional containers from ResourceManager for running its *map* and *reduce* tasks (2). Finally, the AM contacts NodeManager to request the negotiated containers (3) and the NodeManager allocates them and spawns the corresponding YarnChild processes (4).

To avoid cross-applications deadlocks—*i.e.*, when a large number of cluster resources is occupied entirely by containers running MRAppMaster—YARN has introduced the MARP configuration parameter[2]. Concretely, it sets the maximum percentage of cluster resources allocated specifically to MRAppMaster. Since currently only memory is supported, it constrains the amount of memory available to MRAppMaster. MARP also indirectly controls the number of concurrently running jobs and therefore affects the job parallelism and throughput in Hadoop clusters. In summary, YARN has two primary tasks: *Cluster Management* and *Job Scheduling*.

- *Cluster Management* is responsible to monitor the whole Hadoop cluster and to

[2] `yarn.scheduler.capacity.maximum-am-resource-percent`

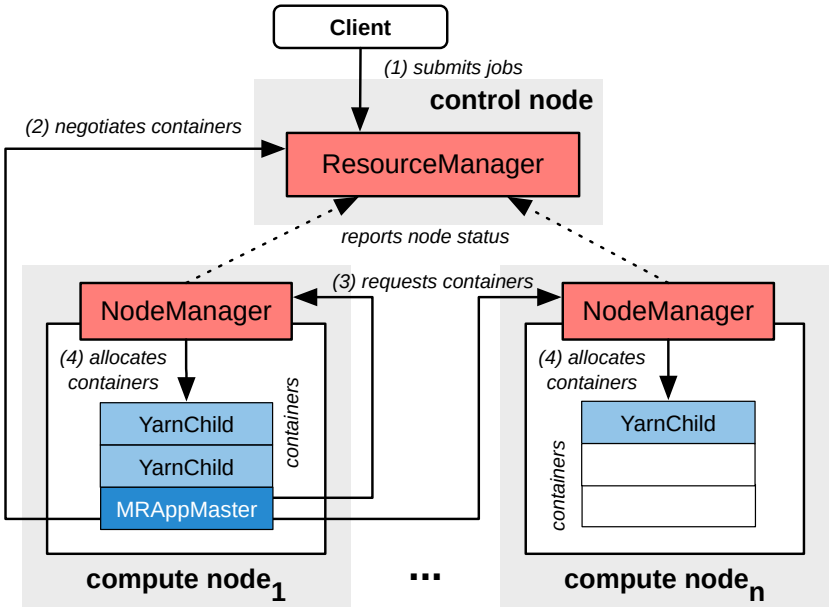


Figure 3.2: High-level YARN architecture.

allocate resources for running jobs. It contains ResourceManager and NodeManager two components, which are depicted in red in Figure 3.2.

- *Job Scheduling* consists of two abstract notions of *containers*: MRAppMaster and YarnChild. They are represented in blue. MRAppMaster is a per-job private controller to request resources for its YarnChild, and to monitor the job process. It is the first *container* of the job, which is launched by ResourceManager and it remains alive until the job has been processed. The latter is a resource block to process map and reduce tasks. Its lifecycle depends only on its assigned task. The different lifecycles of MRAppMaster and YarnChild might cause **Large Drops of Memory Utilization**, which disturbs Hadoop performance. This will be proved in section 5.1.2.

3.2 Platform-level Optimizations

Recently, the performance optimization for MapReduce systems and Hadoop cluster has become a main concern in the domain of Big Data. This has resulted in a number of different approaches that aim to improve Hadoop performance. Because of the complexity of Hadoop, these approaches cover a wide range of research domains. Similar to the self-balancing approach which will be introduced in Chap-

ter 5, many researchers focus on the large number of configuration parameters in Hadoop. They have proposed lots of configuration-related approaches, either to adjust the parameter at runtime, or to filter the optimal configurations for ad-hoc requirements, or belonging to a complex hybrid platform (*e.g.*, Starfish [28]). Scalability is also a popular idea to address the performance issue of Hadoop clusters. Some frameworks, such as FAWKES [22], have been realized and have achieved progress. Moreover, there are also many other researches concentrating on job scheduling, network transmission, data geo-location, etc. The rest of this section will present and discuss these works relevant to the self-balancing approach.

3.2.1 Hadoop Configuration Optimization

AROMA [36] is an automatic system that can allocate resources from a heterogeneous cloud and configure Hadoop parameters for the new nodes to achieve the service-quality goals while minimizing incurred cost. The main idea of AROMA is the parallel pre-analysis of submitted job, which is based on machine learning technologies. This approach can be divided into 2 parts: offline resource allocator and online job profiler. Before the job is forwarded to the Hadoop cluster, AROMA analyzes the resource needs (*e.g.*, RAM, CPU, or disks) of this job. Benefiting from this analysis, the offline resource allocator can start up the appropriate amount and types of VMs for the job from heterogeneous Cloud. Meanwhile, with the training phase based on the experience of previous jobs, AROMA begin to analyze the optimal configurations of Hadoop for the jobs by machine learning technologies. Once the VMs are running, it can quickly deploy and configure an Hadoop cluster, and forward the job to the configured cluster. Thanks to the parallel analysis architecture, AROMA is able to expeditiously decide on the infrastructure of the Hadoop cluster, while benefiting from the start period of VMs to achieve the job profiling, which is a time-consuming work. Thanks to the pre-analysis mechanism, AROMA is able to reconfigure and tune the current Hadoop cluster to adapt to the new requests. But, it also has 2 limitations. (1) The VMs in the Cloud require to be provisioned and be pre-installed with the required Hadoop system. That means, these VMs have been created and are always waiting for provisioning. The resources in Cloud are always under occupation. (2) AROMA is similar to

Chen's proposition [14], which works well when considering ad-hoc jobs. However, for concurrent workloads, its overhead will increase significantly.

Changlong *et al.* [37] also propose a self-configuration tool, named AACT, to maintain the performance of an Hadoop cluster by adjusting Hadoop configurations at runtime. AACT is also based on machine learning. It extracts various informations from JobTracker logs, such as job completion time, input data size, and settings of Hadoop parameters, etc. With these collected data and a few configuration parameters, which are extremely associated to the job processing, AACT can quickly analyze the current Hadoop performance, figure out the misconfiguration, and adjust the specific Hadoop parameters to optimal values. However, the adjustment of Hadoop configurations always requires to restart the Hadoop cluster and to reload the new parameter values. For a multi-tenant cluster, it is impossible to frequently restart the Hadoop system when processing concurrent workloads. The restart will lead to the interruptions of the processing jobs, even the crash of services. Moreover, AACT can optimize the Hadoop performance for a specific job but is difficult to ensure the performance satisfying the user requirements. Finally, for parallel requests (or concurrent workloads), the pre-analysis of AACT may add a heavy overhead to the job processing, and the adjustments for diverse jobs are also possible to conflict each others.

The purpose of Starfish [28] is to enable Hadoop users and applications to automatically get good performance throughout the data lifecycle in analytics. Starfish is a multi-level performance-optimizing system based on Hadoop. Its self-tuning mechanism can be separated into three levels: job-level, workflow-level and workload-level. (1) Job-Level: based on machine learning techniques, it figures out the optimal configurations for the workloads. (2) Workflow-Level: to avoid the performance issues caused by unbalanced data layout, starfish proposes a new workflow scheduler to guarantee the datasets will be evenly stored in HDFS. (3) Workload-Level: besides the job- and data-level optimizations, starfish can also tune the infrastructure of Hadoop cluster to improve system performance, such as scalability. Benefited from the multi-level monitoring and adjustments, Starfish is able to collect various kinds of metrics from an Hadoop cluster, and tune the cluster in various aspects (*e.g.*,, Hadoop parameters, job scheduling, and HDFS) to

accelerate the processing of running jobs. It measures the resource consumption of MapReduce jobs, like CPU cycles of different phases and I/O throughput of reading files from HDFS, to estimate the average map execution time. However, starfish also has several inadequacies. The prediction of workloads to tune Hadoop cluster may extremely differ from the actual situation. Reconfiguring Hadoop parameters may require the restart of the whole system, thereby resulting in the crash of Hadoop services. In concurrent case, due to its complex analytic steps, the overhead of job processing will also increase significantly.

Gunther [38] is a search-based approach for Hadoop performance optimization. In Big Data domain, the cost-based models and machine learning are the two most popular methods to handle performance issues. However, the two methods have various shortages. Cost-based model focuses on researching the presentational relationship between cost metrics (*e.g.*, resource consumption) and performance, but ignores the other causes of performance degradation, such as misconfiguration. Machine learning provides wide studies concerning the performance, but its training phase for application-specific environments degrades the compatibility of the proposed solution. Therefore, Gunther—*i.e.*, a search-based approach—is proposed to overcome the inadequacies of the two methods. Through various analysis and experiments of diverse algorithms, it introduces an evolutionary genetic algorithm to identify the impact of parameter settings, achieving near-optimal job performance. But, due to the complexity of the genetic algorithm, identifying an optimal configuration requires Gunther to repeat computing, thus causing the performance to degrade.

Many other researches focusing on dynamic configurations like [47, 48, 59] also exist. Authors design self-adaptive models to optimize system performance, but their compatibility needs to be reconsidered for YARN.

3.2.2 Scalability at Runtime

Ghit *et al.* [22] have investigated a multi-allocation policies design, FAWKES, which can balance the distribution of hosts among several private clusters. The main idea of this research is to share the infrastructure among multi-tenants. For each tenant, it creates a private Hadoop cluster to process its requests. In this

case, FAWKES is focused on the dynamic redistribution of compute nodes between several clusters while the sum of compute nodes is fixed. The core of this mechanism combines three weighting policies: demand-based weighting, usage-based weighting and performance-based weighting. (1) Demand-based weighting policy depends on the workloads submitted by clients. In accordance with the number of concurrent running jobs in each cluster, this policy balances the distribution of compute nodes. (2) Usage-based weighting policy focus on the resource utilization of each cluster. And (3) the last one (Performance-based weighting policy) is based on 3 statistics collected by MapReduce framework: Job Slowdown, Job Throughput and Task Throughput. Thanks to the cooperation of three weighting policies, FAWKES is able to achieve a good dynamic balance among the clusters, which depends on various statistics. Benefiting from this framework, each MapReduce cluster can have a relatively reasonable performance. However, due to the strict isolation between users, the clusters need to frequently grow or shrink to balance the scales, thereby penalizing each cluster, especially in the case that the workloads are frequent and violent. Furthermore, when the number of tenant become large, FAWKES needs to simultaneously manage lots of private clusters. Therefore, the efficiency of this framework also may become questionable.

Chen *et al.* [14] propose a resource-time-cost model, which can display the relationship among execution time, input data, available system resource and the complexity of Reduce function for an ad-hoc MapReduce job. This model is a combination of the white-box [27] and machine-learning approaches. Its main purpose is to identify the relationship between the amount of resources and the job characteristics. Hadoop clusters can therefore benefit from this research to determine the optimal resource provision strategy, thereby optimizing resource provisioning, while minimizing the monetary cost. However, the limitation of this research is also significant: this model can only work for ad-hoc Hadoop jobs. Typically, to create an on-demand private Hadoop cluster for a specific long-term Hadoop job, this model is an appropriate choice. However, it is not suitable for the concurrent workloads, due to its complex pre-analysis.

Finally, Berekmeri *et al.* [8] introduce a *Proportional-Integral* (PI) controller to dynamically enlist and discharge existing compute nodes from live Hadoop

cluster in order to meet a given target service-level objectives. The core of this approach is a dynamic performance model based on number of nodes and clients. Benefiting from the prediction of the average job completion time, the approach can resize the Hadoop cluster to maintain the system performance. Therefore, how to improve the accuracy of performance prediction becomes the most important issue for this PI controller. To improve the prediction result, this controller combines *Feedback* and *Feedforward Control Loops* together. The feedback loop has two responsibilities: (1) capture the job completion time (discrete or individual performance) of each compute node in Hadoop cluster, (2) counteract the unmeasured and unmodeled disturbances affecting MapReduce system. The feedforward loop measures the disturbance of clients and predict the completion time of each request in various compute nodes. Thanks to the discrete performance (*i.e.*, the job completion time of each compute node) generated by two control loops, the dynamic performance model is able to correctly predict the average job completion time of the whole Hadoop cluster, thereby tuning the infrastructure of Hadoop cluster to improve performance. Nevertheless, this PI controller also has several obvious limitations: (1) its performance model does not take care of the actual resource utilization in Hadoop cluster, but is just based on the number of nodes and clients. When the degradation of performance is caused by other issues (*e.g.*, misconfiguration), provisioning resources will increase unnecessary expenditure rather than efficiently solve the problem. (2) because of the diversity of requests, even though in idle case, the completion time of various jobs may have a big difference. The average job completion time is not able to present the discrete performance in various compute nodes. (3) the PI controller only considers the delay to commit (or decommit) compute nodes into (or from) the Hadoop Cluster. That means, during the whole process, all compute nodes (committed or uncommitted) are always on running and waiting for commission. This results in many extra problems, such as resource waste, energy drains, etc.

3.2.3 Other Optimization Approaches

Some other studies look beyond Hadoop configuration optimization and scalability to library extensions and runtime improvements.

FMEM [60] is a *Fine-grained Memory Estimator for MapReduce jobs* to help both users and the framework to analyze, predict and optimize memory usage. iShuffle [23] decouples shuffle-phase from reduce tasks and converts it into a platform service. It can proactively push map output data to nodes via a novel *shuffle-on-write* operation and flexibly schedule reduce tasks considering workload balance to reduce MapReduce job completion time. Seokyong *et al.* [30] propose an approach to eliminate fruitless data items, as early as possible, to save I/O throughput and network bandwidth, thus accelerating the MapReduce data processing. Benjamin *et al.* [25] deal with a geo-distributed MapReduce system by a two-pronged approach, which provide high-level insights and corresponding *cross-phase* optimization techniques, to minimize the impact of data geo-location. Manimal [33] performs static analysis of Hadoop programs and deploys optimizations, including B-tree indexing, to avoid reads of unneeded data. Panacea [40] is a domain-specific compiler, which performs source-to-source transformations for jobs to reduce the synchronization overhead of iterative jobs. Twister [19] introduces a new in-memory MapReduce library to improve the performance of iterative jobs. Some researches like [50, 65] propose new MapReduce task schedulers to improve resource utilization while observing job completion time goals.

Discussion. The contribution of the self-balancing approach, which will be presented in Chapter 5, complements all these approaches in order to optimize the resource consumption of compute nodes in Hadoop cluster and reduce the completion time of Hadoop jobs.

3.3 Rapid Deployment of Hadoop Cluster

Hadoop is a well-establish data processing framework that is being used extensively in both academia and industry. As such, there exists many commercial and open-source approaches for its provisioning. Nearly each cloud provider provides some support for assembling an Hadoop cluster within its infrastructure, ranging from detailed step-by-step tutorial to fully-featured web applications. The major deployment tools, such as Ansible[3], Puppet[4] or Chef[5], also contain a set of provi-

[3] `http://ansible.com`
[4] `http://puppetlabs.com`
[5] `http://chef.io`

sioning scripts for creating Hadoop deployment. Finally, Cloudera Manager[6] and Ambari[7] are the two most popular tools dedicated for provisioning, managing, and monitoring Hadoop clusters.

Despite that there exists a number of tools that automate Hadoop deployment, they all focus on provisioning long-term production-ready operating clusters. Instead, our approach aims at deployment of experimental environments that are used to rapidly evaluate different Hadoop configurations and adaptation approaches. Furthermore, the existing approaches are usual tight to a particular Hadoop version or configuration, or they bring a large stack of other dependencies. While this is useful for production-ready clusters, it slows down the deployment and complicates experimentations. Finally, a reproduction of an experiment from a cluster provisioned by one of these tools in another environment is again hindered. It requires an access to the particular tools and some levels of knowledge on how to reconfigure it to fit the needs of the target environment.

3.4 Synthesis

Based on the related work enumerated in this chapter, we can easily observe that Hadoop-related researches cover a wide range of research domains. However, the majority of these researches do not concern the issue of job parallelism and throughput of concurrent workloads in Hadoop clusters, particularly in the case of heterogeneous and time-varying workloads. Therefore, I believe that our research can be regarded as a complement to most of above researches. Although some of them also focus on the job scheduling, their proposed solutions are only based on the previous version of Hadoop, but not on YARN. The compatibility of these researches are questionable. For the rapid deployment tool of Hadoop cluster, the idea of our research is to provide a rapid and easy method to deploy a reproducible short-term Hadoop cluster. This leads to that the tool might be interesting to the academic researches of Hadoop, rather than the production. But we believe that the rapid deployment idea based on Docker container technology should be also useful for the industry.

[6] `http://cloudera.com/products/cloudera-manager.html`
[7] `https://ambari.apache.org`

Part : Contribution

Chapter 4

Resource Management in OpenStack

Since several years, Cloud computing keeps conveying the image of a pool of unlimited virtual resources that can be flexibly provisioned to accommodate the user requirements [42]. It provides diverse services across multiple layers—*i.e.*, Software-, Platform-, Infrastructure-as-a-Service—and therefore attracts attention from various business and domains, such as infrastructure maintenance and application development. In this case, more and more Cloud users (and potential users) tend to transfer their business to Cloud computing. Due to the rapid growth in the number of Cloud users, the total resource needs are prone to reach the Cloud infrastructure limitation, thereby threatening the flexibility of resource allocation in Cloud computing. When the Cloud infrastructure has to face such large requirements and falls into congestion degrading Cloud QoS, most of Cloud providers usually think about provisioning physical resources at the infrastructure level to keep the pace of user requests, such physical resources refer to compute nodes which are used to host virtual machines. In this context, many Cloud providers require an automatic manner to dynamically scale Cloud infrastructure according to the dynamics of user requirements, both to ensure the QoS of Cloud computing while avoiding unnecessary expenditure. In this specific case, Cloud elasticity [1, 2, 9, 10, 16, 18, 26, 43, 44, 53, 61, 66] is broadly considered as the *de facto* solution to cope with this problem. However, these elasticity researches usually do not consider the budget limitation of Cloud providers. Even though one of the objectives of these elasticity researches is to minimize the cost of Cloud computing, their premise is that QoS has to reach a reasonable level. Therefore,

such approaches will not stop provisioning resources until the targets have been achieved. However, these elastic approaches can be easily broken by budget requirements and physical limitations (*e.g.*, the lack of servers), particularly in the case of a private Cloud.

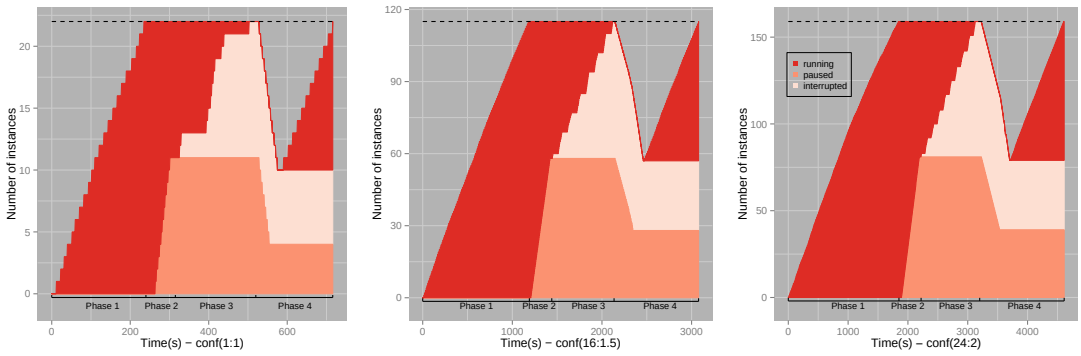
A Cloud platform is generally a huge project to deploy (*e.g.*, OpenStack), which consists of diverse components combined in a complex architecture. In the case of OpenStack, each component can be regarded as an individual system to realize a specific service, such as *Keystone* for providing authentication service and *Glance* for managing images. The architecture of Cloud platform is generally based on simple master-slave mode, but the coordination among the components to achieve a common goal (*i.e.*, Cloud computing) makes the architecture complex and is prone to produce various problems causing diverse impacts. Therefore, inside the Cloud platform, there are plenty of problems that can affect the resource allocation, thus influencing flexibility, such as misconfiguration, idle resource occupation, network loss, and quotas overflow, etc. These problems can be addressed in simple manners (*e.g.*, dynamic configuration) and without any additional cost. In this case, we believe that Cloud elasticity should not be systematically adopted as the primary solution to handle the loss of flexibility—especially if the problem is caused by Cloud infrastructure limitations. Furthermore, the elastic approaches in Cloud computing might lead to a waste of resources that cannot only induce economic losses for the end-users, but also unnecessary carbon emissions for the Cloud providers by over-provisioning the underlying infrastructure.

In Cloud computing domain, OpenStack is an important and well-known open-source project, which consists of a set of sub-systems to constitute a private Cloud across multiple machines. The OpenStack Cloud is designed in a master-slave architecture. This leads to the Cloud providers can easily and quickly provision a new physical host into the Cloud by installing the per-node slave components of the sub-systems. That means, when the Cloud computing cannot support the end-user requirements, that Cloud providers are able to elevate the capacity of the Cloud by scaling the underlying infrastructure. Meanwhile, Cloud providers must pay more for a larger infrastructure, such as energy consumption, carbon emission, and material cost. The growth of Cloud cost will either bothers Cloud providers

by increasing budget, or finally reflects to Cloud user expenditure. However, the insufficient capacity of Cloud computing is not necessarily caused by the lack of Cloud infrastructure. Thanks to IaaS, Cloud users can flexibly allocate resources from the Cloud infrastructure on demand, but this service does not automatically release these resources until the VMs are manually deleted by their owners— *i.e.*, the suspended and inactive VMs in the Cloud will continuously and aimlessly occupy the idle resources which are not available to the other users. This leads to the resource occupation in the Cloud may easily reaches the Cloud infrastructure limitations, and results in the insufficient capacity of Cloud computing. In this case, scaling the Cloud infrastructure is not a preferable choice, compared to the potential of recycling the idle resource occupation. Referring to previous discussions, the resource utilization of Cloud infrastructure becomes a central issue in this thesis.

The main purpose of this research therefore consists in (1) studying the causes of resource leaks, which reduce the actual utilization of Cloud infrastructure—*i.e.*, degrade Cloud flexibility; (2) striving to identify a better solution than elasticity, which does not require any or only needs a little of additional work and cost within an acceptable range; and (3) defining new middleware services in order to support the Cloud infrastructure utilization by implementing smart resource management heuristics atop of existing services. The new middleware aims at improving both *i)* the quality of experience for end-users and *ii)* the QoS of Cloud computing.

This chapter therefore explores an alternative, yet complementary, solution to the resource leaks by adopting the principles of garbage collection [34, 39] in the context of Cloud computing. The primary idea of the approach is to accurately detect *idle VMs* and to dynamically recycle their resources when the Cloud infrastructure reaches its limitation, while ensuring the accessibility and reproducibility of the recycled VMs. At the moment, based on the Python Client APIs provided by OpenStack, this approach (named CloudGC) is implemented as a dedicated solution integrated within OpenStack Cloud. The assessments demonstrate its capacity to timely reduce the waste of resources in an OpenStack Cloud when the user requirements reach the Cloud infrastructure limitation. In addition to periodically and automatically recycle *idle VMs* according to the user needs,



(a) vCPU limitation of straight (b) vRAM limitation of standard (c) vRAM limitation of overcommit

Figure 4.1: Observation of the IaaS limitations on the number of VM instances that can be provisioned.

CloudGC is also able to recover them whenever needed. This ability benefits from a new type of VM instance (*i.e.*, ghost instances) proposed by CloudGC, which can make the Cloud system continuously monitor and store the incoming message requiring the recycled VMs despite that they have been deleted from Cloud instance manager. Moreover, CloudGC also provides the other various services, such as pinned VMs. Thanks to CloudGC, Cloud infrastructures can even switch between operational configurations depending on periods of activities.

4.1 Resource Allocation Analysis

Cloud computing provides a model for enabling on-demand access to a shared pool of computational resources, which can be quickly and easily provisioned and released upon requirements. In the case of *Infrastructure-as-a-Service* (IaaS), these resources take the form of *Virtual Machines* (VMs), which can be created, suspended and deleted by the end-user at any time. Beyond the control of a VM lifecycle, some IaaS solutions, like OpenStack[1], can also control the CPU and memory consumption of VMs though the definition of VM profiles, also known as *flavors*[2] (*e.g.*, tiny, small). Furthermore, the number of VM *instances* can be constrained by the definition of *quotas*[3], which limits the amount of allocable

[1] `https://www.openstack.org`
[2] `http://docs.openstack.org/openstack-ops/content/flavors.html`
[3] `http://docs.openstack.org/openstack-ops/content/projects_users.html`

resources for a given user. While quotas can be used to guarantee that an end-user will not allocate more VMs than allowed, the rest of this section will demonstrate that an IaaS may also suffer from internal constraints that limits its scalability.

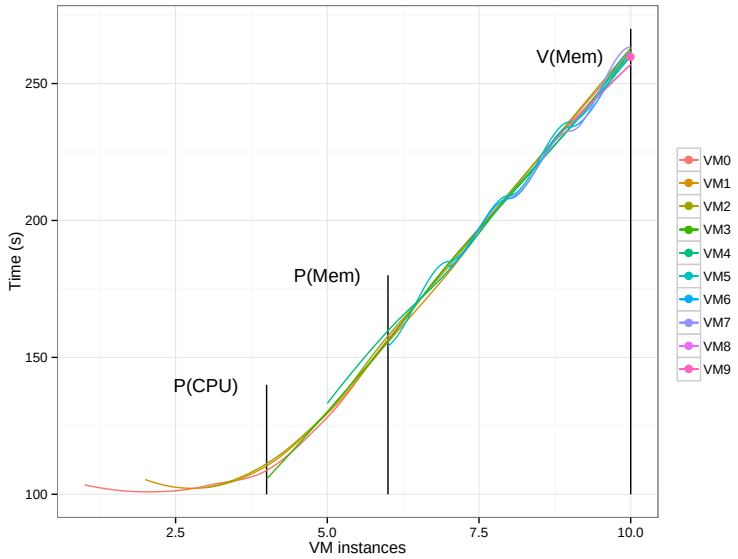


Figure 4.2: Impact of resource overcommitment on VM performance.

To better understand such constraints, a first experiment is run on a vanilla installation of an OpenStack IaaS infrastructure (version 2015.1.4*Kilo*). The hardware setup of testbed is composed of 8 compute nodes, federating 22 CPU cores and 42.2 GB of memory. Each compute node runs Ubuntu (version 15.04) as the operating system, with QEMU (version 2.0.0) as the default hypervisor. The motivation scenario consists in incrementally provisioning new VMs for a single user whose quotas is not fixed—*i.e.*, its quotas are much larger than the Cloud infrastructure limitation. The greedy scenario therefore starts by provisioning new tiny VM instances as long as OpenStack allows it (Phase 1). Once the maximum number of deployed VM instances is reached, the script switches half of the running instances to *pause* state, before trying to provision some additional VM instances (Phase 2). Once this step is completed, the experiment continues to interrupt the other half of running VM instances and tries to provision some more VM instances, again (Phase 3). Finally, the scenario concludes by deleting half of the suspended VM instances—*i.e.*, half of *pause* VMs as well as half of *interrupted* VMs—and allocates new VMs from there until it reaches the infrastructure limitation (Phase 4).

Figure 4.1 reports on the results of executing this experiment for three configurations of OpenStack (straight, standard, and over-commit, detailed in Table 4.1). The standard configuration is the default configuration of OpenStack and maps 1 CPU core to 16 vCPUs and 1 GB of RAM to 1.5 GB of vRAM. In theory, using the standard configuration, the end-user could therefore expect to provision to up to 352 tiny VM instances according to the amount of vCPUs (as each tiny VM requires 1 vCPU and 0.5 GB of vRAM). But, in practice, we note that no more than 115 tiny VM instances can effectively be created by OpenStack, due to the limited resources available. Indeed, in the case of the standard configuration hosting exclusively tiny VMs, 352 ($\frac{352}{1}$) vCPUs can be allocated. However, based on vRAM, the standard configuration only admits 63.3 GB vRAM available in total. That means, for the standard configuration of OpenStack, the vRAM can only support 118 instances ($\frac{63.3-8\times0.5}{0.5}$ as illustrated in Figure 4.1b). The maximum number of tiny VMs is therefore limited to 118 ($\min(\frac{352}{1}, \frac{63.3-8\times0.5}{0.5}) = 118$). The subtraction of 8×0.5 in vRAM is due to the virtual memory reserved by the OpenStack components deployed in all compute nodes—*i.e.*, the testbed contains 8 compute nodes and each node requires 0.5 GB vRAM for running OpenStack. This rule can be proven in straight and over-commit configuration. Using straight configuration as example, the amount of vCPUs is 22 and OpenStack supports 42.2 GB vRAM. The maximum number of tiny VMs for straight configuration should be $\min(\frac{22}{1}, \frac{42.2-8\times0.5}{0.5}) = 22$ which can be observed in Figure 4.1a.

Table 4.1: Overcommit ratios used as configurations.

configuration	mapping	vCPUs (total)	vRAM (total in GB)
straight	1:1	$22 \times 1 = 22$	$42.2 \times 1 = 42.2$
standard	16:1.5	$22 \times 16 = 352$	$42.2 \times 1.5 = 63.3$
over-commit	24:2	$22 \times 24 = 528$	$42.2 \times 2 = 84.4$

By increasing the ratio of vCPUs and vRAM (cf. Figure 4.1c), one can observe that the number of deployed VM instances can be raised, but over-committing CPU and RAM resources to increase the capacity is not a free lunch—*i.e.*, it has a cost for the Cloud. In order to demonstrate the impact of resource overcommitment, we compare the system performances of each provisioned VM in different moments. We log the completion time of a benchmark running continuously in each of the VM instances we provisioned in the Cloud. In particular,

we use CPU test provided by the SysBench benchmark[4]. Figure 4.2 reports on the evolution of this completion time per VM when provisioning from 1 to 10 new VM instances on a single compute node, using the standard configuration of OpenStack. One can observe that, once the number of allocated physical core is reached on a compute node (P(CPU)), the benchmark performance becomes linearly impacted by the provisioning of new VMs: the more VM instances are provisioned, the longer time each VM takes to complete the benchmark.

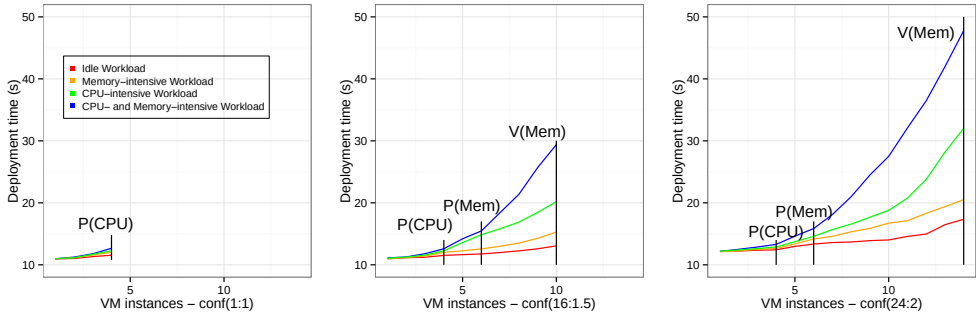


Figure 4.3: Impact of overcommitment on VM deployment.

Figure 4.3 compares the deployment time of one tiny VM and exposes the impact of over-commitment—*i.e.*, standard configuration also has a weak over-commitment compared to straight configuration—on the VM deployment delay in different environments. This experiment deploys the VMs one by one until OpenStack prohibits a new deployment. Along the execution of this scenario, each provisioned (deployed) VM continuously processes a given workload to consume the specified resources in order to simulate different deployment environments (idle, CPU intensive, RAM intensive, and CPU-RAM intensive). Meanwhile, the deployment time per new VM is recorded while the other provisioned ones process different workloads (*e.g.*, CPU-RAM intensive). To observe the universality of the phenomenon appearing on deployment time, this experiment is repeated in the 3 configurations on the same single compute node. With different configurations (*e.g.*, straight, standard, and over-commit) and deployment environments, the deployment time of new VM instance exposes a similar behavior like the completion time of SysBench reported in Figure 4.2. The more VMs are deployed in Cloud, the longer time the deployment of one new VM takes. Therefore, although it raises

[4]`http://manpages.ubuntu.com/manpages/xenial/man1/sysbench.1.html`

the limits of the Cloud infrastructure capacity, resource over-commitment has to be wisely tuned by the Cloud provider in order to limit the performance impact for the end-user. Furthermore, changing the OpenStack parameter for adjusting the resource over-commitment requires the nova sub-component (named nova-scheduler) to be restarted. This inevitably leads to the interruption of resource allocations in the whole OpenStack Cloud. Given that the two sets of experiments all shows the negative impact of resource over-commitment on diverse performances (Cloud resource allocation performance and VM processing performance) and the limitation of reloading OpenStack parameters, most of Cloud providers investigate alternative solutions to increase their resource availability atop of their current Cloud infrastructure.

As initially observed in Figure 4.1, no matter its level of activities, the only solution to release the resources occupied by the VM instances is to delete these instances. The first experiment reveals this solution before provisioning new VMs (Phase 4). Therefore, by recycling the resources provisioned—but not actually used—by VM instances, one can expect to obtain available resources from a Cloud infrastructure without systematically falling back on Cloud elasticity techniques, thereby raising the capacity of this Cloud.

The challenge to be tackled in this context therefore consists in automatically recycling the *provisioned but not used* resources—*i.e.*, *idle* VM instances [35]. To recycle the VM instances in order to raise the capacity of Cloud infrastructure, this research must overcome 2 obstacles:

– The first obstacle is to automatically and accurately detect *idle* VM instances—*i.e.*, the VM instances which are provisioned, but not actively used—in order to recycle them. In OpenStack Cloud, provisioned VM instances have different states. Some of them can be easily defined as *idle*, but some others are prone to be confused with actively used instances because they have the same state in OpenStack instance manager. We distinguish between two categories of *idle* VMs: (1) instances which are *explicitly inactive* (*e.g.*, manually paused or interrupted) and (2) instances which are *implicitly inactive* (*e.g.*, having running state but no CPU activity for the last 10 hours).

– The second obstacle is to automatically recover the recycled VMs, once end-users require them again. As the *idle* VMs will be recycled by a third-party component (*e.g.*, the contribution of this research) and not the VM owners, these recycled VMs should be kept alive in a different form, so that end-users may request them again. However, in an OpenStack Cloud, when a VM instance is deleted, it is completely removed from the Cloud instance manager. At the moment, any new external request requiring the recycled VMs cannot succeed even only *ping* to them. And then, in this case, how to automatically recover the recycled VM instances whenever needed become an issue in this research.

Based on this assumption, the remainder of this chapter introduces a new middleware solution to recycle the *idle* VM instances. Given that a Cloud provider or a Cloud administrator cannot manually manage a potentially large population of VM instances, she requires a middleware service that can take care of resource management duties in order to ensure that resources are always made available to the end-users. Our approach can therefore be symbiotically integrated upstream of an elasticity service, thus ensuring that these services are only activated when all the Cloud resources are allocated and actively used.

4.2 CloudGC Middleware Service

To seamlessly integrate the VM recycling mechanism with a Cloud infrastructure, this solution is designed as a middleware service that can interact with the existing services made available by an OpenStack IaaS. Our middleware solution, named CloudGC, is therefore inspired by the garbage collection mechanism that is embedded in virtual machines like Java and used to reclaim the memory occupied by objects that are no longer in use by the applications. During the last three decades, garbage collection has focused a lot of research activities, moving from *"stop-the-world"* algorithms to generational approaches [4, 34]. This form of automatic memory management approach periodically scans the memory of the virtual machines (*e.g.*, JVM) and collects *garbage objects* to free the associated memory in order to facilitate the allocation of new objects.

CloudGC therefore builds on the results achieved in garbage-collected languages in order to apply the principles of garbage collection technologies on VM management and at the scale of the OpenStack Cloud. However, unlike the abandoned objects in applications of Java, *idle VM instances* in OpenStackCloud are software artifacts that might still be used in the future. Therefore, managing the occupation of idle resources in a Cloud infrastructure does not only focus on recycling the *idle* VM instances, but it also needs to handle the recovery of the recycled VM instances. In this context, the challenges of developing a garbage collector for the Cloud are threefold, including *i)* to automatically and accurately detect idle VM instances, *ii)* to efficiently recycle the VM resources, and *iii)* to support the automatic recovery of recycled VM instances.

The following sections address each of these challenges more specifically, as well as their impact on the lifecycle of a VM in an OpenStack Cloud.

4.2.1 Lifecycle of VM Instances

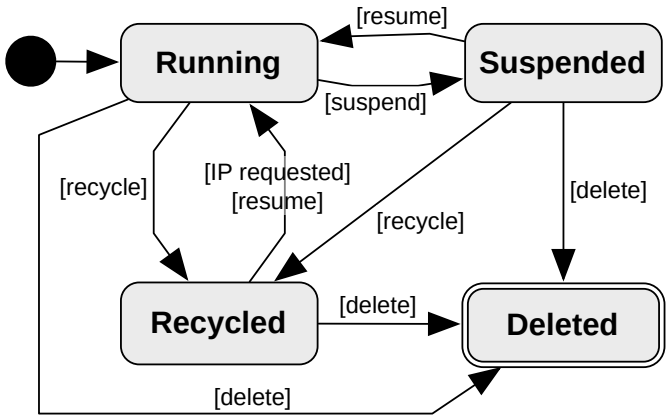


Figure 4.4: Lifecycle of a VM instance in CloudGC.

Figure 4.4 depicts the lifecycle of a VM instance in OpenStack IaaS. There are three types of standard states for the lifecycle: running, suspended (*e.g.*, paused and interrupted), and deleted. The transitions among the standard states is manual. In this case, end-users have to manage the VM instances, manually. With a large population of VM instances, this is prone to leave some instances become *idle*, that may occupy resources but not use them, due to oblivion or negligence of VM owners.

Beyond the standard lifecycle of VM instances proposed by OpenStack, CloudGC adds a new state recycled, which corresponds to the transitory state that a VM takes when it is recycled by the middleware service. Thanks to CloudGC, a VM instance can automatically become recycled from running and suspended states when (1) Cloud infrastructure cannot support any more new user requirements **and** (2) this VM instance has been determined as *idle* one by CloudGC. A VM instance can also leaves this state by diverse methods—*i.e.*, either by being manually resumed (in the case of a suspended VM), accessed on-demand (in the case of a running VM), or deleted manually as shown in Figure 4.4. Whether a VM instance is in the running or suspended stage, CloudGC can accurately determine the idle one, and therefore decide to recycle the enough *idle* VM instances for the new requirements, as further explained below.

The following 3 sections introduce the steps implemented by CloudGC to move a VM from a running or suspended state to the recycled state, back and forth.

4.2.2 Detecting Idle VM Instances

CloudGC builds on the assumption that not all the VM instances are continuously used in a Cloud infrastructure, which cannot be considered as unlimited—*i.e.*, the Cloud infrastructure limitation can be easily reached, *e.g.*, in a private Cloud. Therefore, a critical part of the VM recycling process implemented by CloudGC consists in detecting VM instances that are considered as *idle*, which aims at (1) providing candidates for the following steps of recycling process and (2) avoiding CloudGC recycling active VM instances thus degrading the QoS of Cloud but should originally improve it. Idle VM instances have two categories: either VM instances that have been explicitly suspended by the end-users (*e.g.*, a VM in the paused or interrupted state) or VM instances that have not been active (*e.g.*, no CPU activity) for a long period of time in running state. CloudGC distinguishes between *explicit* and *implicit idle* VM instances: the former is not intended to be used by the end-user on a short-term basis, while the latter might be triggered at any time. Therefore, to avoid the frequently repeated snapshots (which will be used by recovering step) of *idle* VM instances, CloudGC only

backs up the *explicit idle* ones. This does not only reduce the network load of snapshots, but also significantly shortens the time of the detection step. Nevertheless, inspired by generational garbage collectors [57], we assume that the longer time a VM has been flagged as inactive in the past, the longer time it will still be idle in the future. In this case, all the determined *idle* VM instances will be ordered by their inactivity durations to make sure that the *idle* VM instances will be recycled firstly.

Due to the diverse types of *idle* VM instances, CloudGC therefore maintains two queues to record the determined *idle* instances: the *explicit queue* and the *implicit queue*. To detect and track idle VM instances, CloudGC periodically synchronizes the list of deployed VM instances from the IaaS instance manager (*i.e.*, a **Nova** service). The list of suspended VM instances, ordered by interruption dates (oldest first), is used to generate the explicit queue. For the elements in explicit queue, when they do not exist in the suspended list anymore, their metadata as well as snapshots will be removed from the explicit queue and the IaaS storage service (*i.e.*, a **Glance** service). Then, for the VM instances that are in the suspended list but not in explicit queue, CloudGC records them in the explicit queue and immediately back up them. From the list of active VM instances, CloudGC queries the IaaS monitoring service to filter out the VM instances whose CPU activity has not exceeded a given threshold for a given duration (the CPU activity threshold and the duration are two configuration parameters of CloudGC to control the level of garbage collection). And then, CloudGC compares the selected VMs from the active list and the items from the implicit queue. The implicit queue updates in 2 steps. (1) The items from the implicit queue that are not in the selected VMs are firstly removed, before (2) inserting the selected VMs that are not in the implicit queue. The output of this first phase therefore delivers two lists of idle VM instances (explicit and implicit), ordered by inactivity durations.

4.2.3 Recycling Idle VM Instances

As previously mentioned, unlike objects in garbage collected languages, the recycled VMs instances can be recovered upon user requests. Therefore, recycling

the *idle* VM instances does not only consist in releasing the Cloud resources that are associated to each of the instances, but it also requires to save the current state of the VM instances in order to be able to recover them in a similar state, if necessary. In CloudGC, the state of idle VM instances is saved as a snapshot in the IaaS storage service. For the snapshots of *idle* VM instances that are already stored in the IaaS, they are automatically overridden by CloudGC if some activity has been detected since the last version. CloudGC only automatically builds a snapshot of explicit idle instances when they are inserted into the explicit queue in detecting step. Given that the activity of implicit idle instances is not frozen, this may lead to frequently overriding the snapshots, thus increasing the network load and delaying the detection step, as above explained. CloudGC can only build a snapshot of an implicit idle instance on-demand—*i.e.*, when the VM instances requires to be recycled. Recycling explicit *idle* VM instances is therefore much faster than recycling of implicit ones.

CloudGC defines the *idle* VM queues with different priorities. Given that the items in the explicit queue have been snapshotted, recycling them only spends a short time to delete the VM from IaaS instance manager and does not need to wait for snapshotting the state of the VM instances. This saves a lot of time to free the resources associated to the recycled VMs and thereby guarantees the QoS of CloudGC. Then, upon the exhaustion of the explicit queue, the items in the implicit queue are elected for recycling in order to satisfy user requirements, if needed. Benefiting from the priorities, CloudGC can ensure that the explicit idle VMs are recycled in priority compared to implicit ones.

Corresponding to recycled state, CloudGC introduces a new mechanism, which maintains the IP address of recycled VMs available to expose them as alive from the perspective of users. Thanks to this mechanism, a recycled VM can be still available to third parties, even though they have been removed from the Cloud. In this case, the recycled VM can be regarded as a *ghost instance*. In both cases of recycling explicit or implicit *idle* VM instances, CloudGC uses the IaaS instance manager to rebind the IP address of the *idle* VM instance to a *ghost instance*, which acts as a proxy to recover the VM upon user requests from a third party. Upon completion of the VM snapshot, the instance is deleted from the IaaS

instance manager, thus effectively releasing the associated resources. While this process can be applied to recycle all the detected idle instances, CloudGC takes the amount of resources to be released as an input, based on the number and the flavors of the new VM instances to be provisioned. Thus, CloudGC only recycles the necessary idle instances to allow the IaaS to provision the new requested VM instances. This avoids unnecessary recycling process, also shortens the time of recycling step, and leaves other *idle* VM instance alive (which may be required at any time), thereby improving the QoS of CloudGC. If the recycling process fails to release the requested resources, the Cloud infrastructure can either reject the incoming provisioning request, or trigger an elasticity service to provision some additional compute nodes, thus increasing the capacity of the IaaS and improving the effect of elasticity service by avoiding over-provisioning resources.

4.2.4 Recovering Recycled VMs

CloudGC recycles idle VM instances to ease the deployment of new VM instances in order to elevate the resource utilization of Cloud infrastructure. Nevertheless, the recycled VM instances may also be triggered at any time, *e.g.*, by requesting a resource or a service supported by a ghost instance. In such a case, CloudGC should be able to recover the recycled VM associated to the ghost instance in the same state and configuration where it was before being recycled, before forwarding the incoming requests to the VM instance. As part of this recovery process, one can note that provisioning a recycled VM instance may require CloudGC to recycle other idle VM instances for available resources, like an iterative process. Therefore, the recovery process of CloudGC follows the same workflow as for provisioning a new VM instance in OpenStack Cloud, but loading automatically the snapshot as VM image also from OpenStack Image Service (Glance) and restoring the initial VM configuration (*e.g.*, getting the floating IP address from the ghost instance and rebinding it to the associated recovered VM instance).

4.2.5 Other services

Both recycling idle instances and recovering recycled instances are not instantaneous processes, taking from seconds to minutes depending on network bandwidth, the number of snapshots to be stored as well as the amount of resources to be recycled and recovered. Meanwhile, for some VM instances whose services are too important to be interrupted or response time are extremely critical, these VMs cannot be recycled in any cases. To prevent CloudGC from recycling VM instances that are considered as critical (*e.g.*, expected to react as quickly as possible to incoming requests), a VM can also be *pinned* on the Cloud. Pinned VM instances are therefore made invisible from the detection and recycling processes, no matter their activity or their current state.

For many services which are highly time-constrained, their response time are critical only in a specific period, and their resource consumption become idle in the other durations. In this case, the service providers will pay over-high cost for the service infrastructure, while the Cloud also suffers from low periodic resource utilization. Thanks to the recycling and recovering services as well as a time table, CloudGC can also switch the periodic services which focus on different periods and allows them to share the same Cloud infrastructure.

The following section dives into the implementation details of CloudGC as a middleware service integrated in OpenStack.

4.3 Implementation Details

This section dives into the details of the integration of CloudGC within OpenStack Cloud. Thanks to the client libraries provided by OpenStack, CloudGC is able to cooperate with the services supported by other OpenStack components OpenStack is widely considered as the *de facto* OSS standard for deploying a private Cloud which provides an IaaS solution on the top of sets of physical machines, which is the representative of the environments supporting CloudGC in this research.

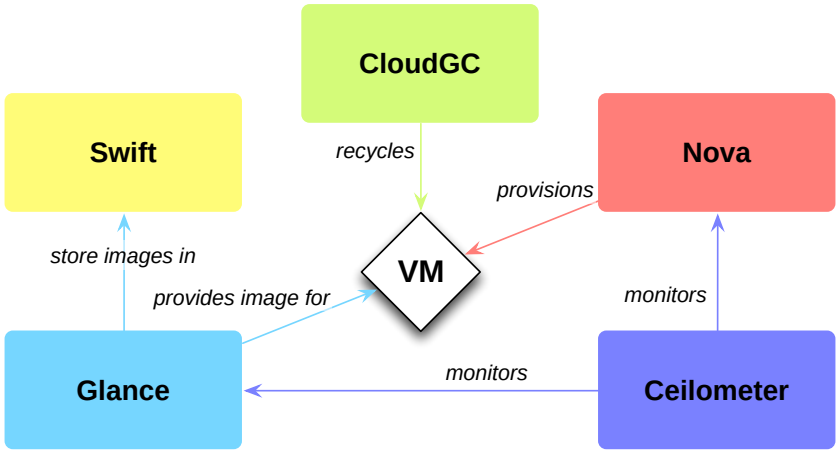


Figure 4.5: Integration of CloudGC in OpenStack.

4.3.1 Middleware Overview

Figure 4.5 depicts the integration of CloudGC in the OpenStack IaaS. This implementation is a Python-based project to seamlessly integrate with OpenStack, which is also implemented in Python. Even though CloudGC is a VM-related middleware, its recycling services rely on various services provided by other different components of OpenStack[5]. Among all the components of OpenStack, CloudGC interacts more specifically with Nova, Ceilometer, Glance, and Swift. CloudGC builds on the standard APIs provided by each of these services to support the VM recycling process. In particular, CloudGC uses Nova (a core component of OpenStack) to recycle idle VM instances and to recover the recycled VM instances (with their configuration), while its Cloud instance manager service does not only help the detection step of the recycling process to update the *idle* VM queues in CloudGC, but it also keeps the floating IP addresses for the ghost instances, which are extremely important for the recovery step. Glance and Swift provide the necessary support to automatically manage, save and restore the snapshots as well as configurations of recycled VMs, respectively. Finally, the monitoring capability of Ceilometer is used by CloudGC to analyze the activity of deployed VM instances. Ceilometer keeps continuously to monitor the CPU activity of each provisioned VM at every moment. Based on the metrics provided by this service, CloudGC can accurately determine and record the *implicit idle* VM instances. Moreover, the standard APIs also provide the possibility and space for

[5] `https://www.openstack.org/software`

future development of CloudGC to adapt to the development of OpenStack.

OpenStack is an open-source project achieved in Python. In this case, this implementation of CloudGC is also implemented in Python, thus benefiting from the client libraries made available for each of these services. By adopting this service-oriented architecture, CloudGC therefore integrates seamlessly within the OpenStack Cloud and the implementation of the recycling process does not affect neither the API of existing services nor the GUI provided by Horizon as well as the administration console of OpenStack.

The architecture of CloudGC is structured in 3 components—Monitoring, Recycling and Recovery—within 2 types (*i.e.*, active and passive), whose details is shown in the following sections.

4.3.2 Monitoring Component

The monitoring component is an active component defined by CloudGC to periodically query Nova for the list of deployed VM instances and to update two shared priority queues, which are used to record the *idle* VM instances—*i.e.*, the idle VMs with longest idle duration are enqueued firstly. Algorithm 1 summarizes the behavior that is periodically executed by this monitoring component (the period can be configured by CloudGC). The monitoring component is launched in an individual thread, which runs in background, to guarantee the maintenance interval of two *idle* VM queues is strictly in compliance with the period defined in CloudGC configuration. It is a periodically active component guarding 3 global lists: *explicit idle queue*, *implicit idle queue*, and one specific list (*pinned list*) to record the pinned VM instances which cannot be recycled in any cases.

As mentioned in Section 4.2.2, CloudGC distinguishes between *explicit* and *implicit idle* VM instances in order to recycle explicit VM instances in priority to minimize the negative impact caused by recycling process. The second level of priority in CloudGC indicates that this middleware service will recycle the running VM instances that have been idle for a while, if needed. Given that the implicit queue is sorted by idleness of each item, the item which is idle for the longest period will be recycled firstly.

Beyond the two idle queues, the monitoring component also contains a passive

Algorithm 1 Monitoring behavior of CloudGC

```
1: global ExplicitQueue
2: global ImplicitQueue
3: global PinnedList
4: procedure Monitoring(duration)
5:     Clear(vms)
6:     Clear(backup)
7:     vms ← list(Nova, ¬ PinnedList)
8:     idles ← Filter(vms, PAUSED|INTERRUPTED)
9:     for vm ∈ ExplicitQueue do
10:        if not Contains(idles, vm) then
11:            Remove(ExplicitQueue, vm)
12:            Delete(Glance, vm)
13:    for vm ∈ idles do
14:        if not Contains(ExplicitQueue, vm) then
15:            Insert(ExplicitQueue, vm)
16:            Add(backup, vm)
17:            Sort(ExplicitQueue)
18:    Snapshot(Glance, backup)                    ▷ snapshot in parallel
19:    actives ← Filter(vms, RUNNING)
20:    for vm ∈ ImplicitQueue do
21:        if not Contains(actives, vm) then
22:            Remove(ImplicitQueue, vm)
23:    for vm ∈ actives do
24:        if Idle(Ceilometer, vm) ⩾ duration then
25:            if not Contains(ImplicitQueue, vm) then
26:                Insert(ImplicitQueue, vm)
27:                Sort(ImplicitQueue)
```

interface (shown in Algorithm 2) to maintain and update another list for *pinned* VM instances. Regardless the actual state of the *pinned* VMs, all the items in this list are ignored by detection function in the monitoring component and therefore never considered as the targets of the recycling process.

4.3.3 Recycling Component

The recycling component is a passive component introduced by CloudGC and triggered by Nova when it fails to allocate resources for an incoming provisioning request. In that case, Nova requests the recycling component of CloudGC to recycle a number of *idle* VM instances which have been already recorded in the two *idle* VM queues, in order to free a sufficient volume of resources to satisfy the provisioning request. Algorithm 3 reports on the implementation of this com-

Algorithm 2 Pin specific VMs in CloudGC

```
1: global ExplicitQueue
2: global ImplicitQueue
3: global PinnedList
4: function Pin(vms)
5:     for vm ∈ vms do
6:         if not Contains(PinnedList, vm) then
7:             Insert(PinnedList, vm)
8:         if vm ∈ ExplicitQueue then
9:             Remove(ExplicitQueue, vm)
10:            Delete(Glance, vm)
11:        if vm ∈ ImplicitQueue then
12:            Remove(ImplicitQueue, vm)
```

ponent within an OpenStackCloud, and also illustrates the recycling priorities of the two *idle* VM queues, which are introduced in CloudGC. The recycling component always recycles the *explicit idle* VM instances firstly until the *explicit* queue becomes empty. Otherwise, any running instance (*i.e.*, *implicit idle* VM instances) will not be recycled. Unless the *explicit* queue is empty and the freed resources are still not enough to meet the new requirement, *implicit idle* VM instances will begin to be considered as candidates to be recycled. but this will delay the completion of recycling process due to the snapshots of *implicit idle* VM instances. If CloudGC succeeds to recycle a sufficient amount of resources, Nova can retry to provision the new VM instances. In case of failure, Nova can reject the request or trigger some horizontal elasticity support for OpenStack, which is out of the scope of this research.

To achieve the recycling process, recycling component contains sets of passive functions. The Store, Rebind, and Delete operations of recycling process save the instance configurations, rebind the VM instances on the ghost instance, and delete all the selected idle VMs at once, respectively. Although the operation Snapshot appears in Algorithm 1 for snapshotting all items in explicit queue, this operation is automatically called by the operation Pause as it is the case for snapshotting implicit idle VM instances on-demand. Furthermore, to reduce the recycling delay, all the selected idle VMs are recycled in parallel.

Algorithm 3 Recycling behavior of CloudGC

```
1: global ExplicitQueue
2: global ImplicitQueue
3: function Recycle(volume)
4:     recycled ← ∅
5:     backup ← ∅
6:     while Available(Nova, recycled) < volume do
7:         if not Empty(ExplicitQueue) then
8:             vm ← Get(ExplicitQueue)
9:             Add(recycled, vm)
10:        else if not Empty(ImplicitQueue) then
11:            vm ← Get(ImplicitQueue)
12:            Add(recycled, vm)
13:            Add(backup, vm)
14:        else
15:            return FAILURE                    ▷ lack of idle VMs
16:    Pause(Nova, backup)
17:    Snapshot(Glance, backup)                  ▷ called by Pause
18:    Store(Swift, recycled)
19:    Rebind(Recovery, recycled)
20:    Delete(Nova, recycled)
21:    return SUCCESS                            ▷ idle VMs recycled
```

4.3.4 Recovery Component

The recovery component is in charge of handling incoming requests that require some recycled VM instances. To do so, CloudGC binds the floating IP of the recycled VM instance to a ghost instance as part of the recycling process (cf. Algorithm 3), so that the recycled VM instances are still perceived as available from outside the OpenStack Cloud, but they have been removed in fact to free resources. Therefore, upon receiving an incoming request requiring an recycled VM, the corresponding ghost instance triggers the recovery function described in Algorithm 4 and then forwards the incoming request to the recovered VM—if the provisioning process succeeds—or returns an error to the end-user. Additionally, for the VM instances that need to run periodically, CloudGC proposes a timer, which acts as `crontab`, to recycle and recover these VMs periodically. This solution ensures that the periodic VM instances are always being scheduled in the Cloud in order to complete their periodic jobs. Thanks to this periodic switch service supported by CloudGC, these VM instances running in different periods can share a same OpenStack Cloud, thus decreasing the VM service cost for ser-

vice providers (*i.e.*, Cloud end-users) as well as improving the resource utilization of Cloud infrastructure.

As already mentioned, the recovering process can be regarded as a provisioning process, except that the VM will be created from a ghost instance. The provisioning process may in turn trigger the recycling process in prior to free enough resources for provisioning the requested VM instance, thus introducing an unpredictable delay for processing the incoming request. The recovering process may also suffer from a such delay. In the case of a saturated Cloud, to recover a recycled VM instance, the recovering process must invoke firstly the recycling step to recycle some *idle* VM instances, then restores the recycled VM instance from the ghost instance. Therefore, the response time of services supported by the recycled VMs will add an extra delay. However, the cost of recovering a recycled VM instance is only paid upon the first incoming request. Moreover, this weakness of CloudGC is further mitigated by the support for *pinned* VM instances, which can be kept alive in OpenStack instance manager—*i.e.*, they will never be recycled even though become idle—to deliver a better response time when a VM instance is considered as critical for the end-user.

Algorithm 4 Recovering behavior of CloudGC

```
1: function Recover(ip)
2:     image ← Retrieve(Glance, ip)
3:     config ← Retrieve(Swift, ip)
4:     vm ← Provision(Nova, image, config)
5:     if vm = NULL then
6:         return FAILURE                    ▷ No more resource available
7:     else
8:         Rebind(Nova, vm, ip)              ▷ Disabling the ghost
9:         Delete(Glance, ip)                ▷ Freeing the storage
10:        Delete(Swift, ip)
11:        return SUCCESS
```

In the next section, our experiments demonstrate how the combination of these three components in CloudGC performs for different scenarios and evaluate the overhead introduced by this new middleware service of OpenStack. The evaluations show that CloudGC does not only help user requirements exceed the Cloud infrastructure limitation, but it can also support various services which are useful in different cases.

4.4 Validation in Private Cloud

The resource leaks caused by a stiff mechanism of resource allocation in OpenStack decrease the resource utilization of Cloud infrastructure. In a saturated Cloud, they also may elasticity solutions to keep provisioning new compute nodes whenever new VM instances needs to be deployed. In this specific case, CloudGC is proposed to cope with the resource leaks, by using the components introduced in previous section. This section assesses CloudGC with regards to the objectives defined in Section 4.1—*i.e.*, elevating the limits of a Cloud infrastructure to stop wasting resources. The assessments therefore report on various scenarios, which are considered to demonstrate the capability of CloudGC. Our results reveal that CloudGC can better manage the resources of OpenStack Cloud than a bare OpenStack IaaS. In this section, the testbed we use is the same hardware infrastructure as in Section 4.1 and OpenStack is configured to run with the standard configuration.

4.4.1 The Sky Is The Limit

In this first experiment, we run a similar scenario to the one described in Section 4.1—*i.e.*, we firstly saturate the Cloud infrastructure, then suspend some VM instances before trying to provision some additional instances. Figure 4.6 depicts the results that CloudGC achieves on such a scenario. In particular, while the number of VM instances that can be provisioned in a standard OpenStack is limited, as emphasized in Figure 4.1b, CloudGC demonstrates its capacity to recycle the idle VM instances to accept the provisioning of new VM instances beyond the limits we previously observed (black dashed line). CloudGC recycles in priority the VM instances that are explicitly paused or interrupted in order to accommodate the incoming provisioning requests.

In details of Figure 4.6, after half of running VM instances becoming paused, the end-users are able to deploy new VM instances—*i.e.*, recycled VM instances begin to appear when new VM instances are deployed (the dark blue part exceeds the limit of Cloud infrastructure). Then, the scenario continues to interrupt the other half of running VM instances and tries to deploy more VM instances. In this case, one can find that, because of the ordered idle VM queues, CloudGC

recycled all paused VMs before interrupted ones. As shown in Figure 4.6, the total number of VM instances, which contains the ghost instances (*i.e.*, the recycled VM instances), can exceed the Cloud infrastructure limitation. This experiment therefore prove that CloudGC can elevate the Cloud capacity to serve more end-users in parallel.

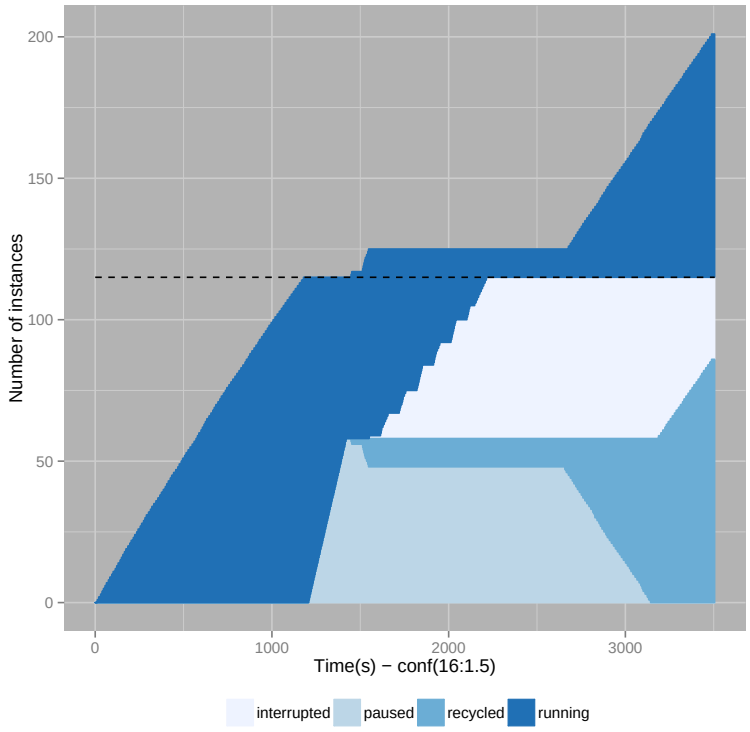
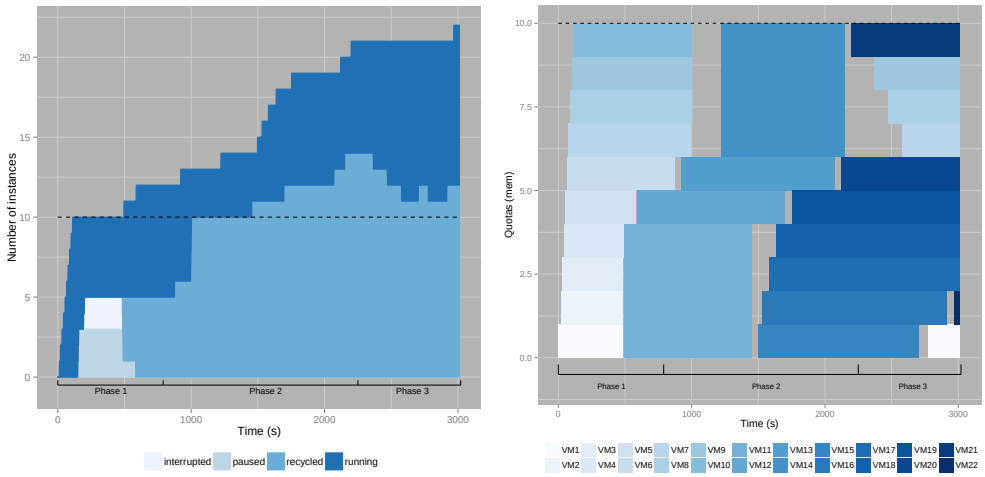


Figure 4.6: Provisioning VM instances with CloudGC.

For the sake of readability, Figure 4.7 zooms in a single compute node of OpenStack Cloud and shows how VM instances are recycled and scheduled by CloudGC along time. In particular, Figure 4.7a demonstrates that once the paused or interrupted VM instances are all recycled (cf. Phase 1), CloudGC then focuses on the running instances (cf. Phase 2). As explained in previous section, CloudGC maintains two idle VM queues: *explicit* and *implicit*. Beside the suspended VM instances, the monitoring component also needs to identify those running instances, which are considered as *idle*. Given that the two queues have different priorities, CloudGC always recycles explicit idle VMs (cf. Phase 1) before implicit idle VMs (cf. Phase 2). Figure 4.7a depicts also in Phase 3 that the recycled VM instances can still be requested at any time, and one can



(a) Evolution of recycled VMs. (b) Scheduling of VMs along time.

Figure 4.7: Node-scale scheduling of VM instances using CloudGC.

observe that CloudGC succeeds in recovering these requested VMs (light blue part shrinks). This recycling process keeps working as long as there are enough idle VM instances that can be recycled to satisfy a new provisioning request.

Beyond the results already reported, Figure 4.7b demonstrates more specifically the capacity of CloudGC to deal with different VM flavors, for example by recycling 4 tiny instances (1 vCPU / 512 MB) to provision a small one (1 vCPU / 2 GB). In Figure 4.7, one can also observe that the provisioning delay of the two small instances defer from each other. The first small instance is quickly provisioned because CloudGC recycles explicitly idle VM instances and does not have to snapshot their state (as the snapshot is operated upon the interruption of the VM instance and takes a long time). However, the provisioning of second small instance requires CloudGC to recycle implicitly idle VM instances, which induces an additional overhead for snapshotting the state of 4 tiny instances. Therefore, one can find that provisioning the second small instance takes much longer time than the first one. Furthermore, at the end of this experiment, the recovery process is also demonstrated in different cases. For some recycled VMs (*e.g.*, $vm1$, $vm7$, $vm8$, $vm9$), one can also find that they are able to be recovered when the Cloud infrastructure has enough resources (cf. $vm7$, $vm8$, $vm9$) or has to launch iterative process to recycle idle VM instances firstly (cf. $vm1$).

Figure 4.8 focuses on another OpenStack compute node and shows how

CloudGC behaves when pinning a VM instance. At the beginning, this scenario deploys 10 VMs as the OpenStack configuration allows them, and directly suspends them. Then, the first deployed VM (*i.e.*, $vm1$) is pinned by end-users. In the rest of this scenario, when end-users begin to deploy new VM instances, CloudGC starts to recycle the explicit idle VMs from $vm2$ and leaves $vm1$ alone. This situation only can be changed after $vm1$ becoming unpinned (released). As expected, when scenario requires to deploy a new VM (*i.e.*, $vm14$), $vm1$ is selected and quickly recycled. In summary, one can assess that the pinned VM instance is not affected by the recycling process of CloudGC, no matter its current state (running or suspended). CloudGC only recycles the VM instances that are considered as *recyclable* (*i.e.*, *unpinned*).

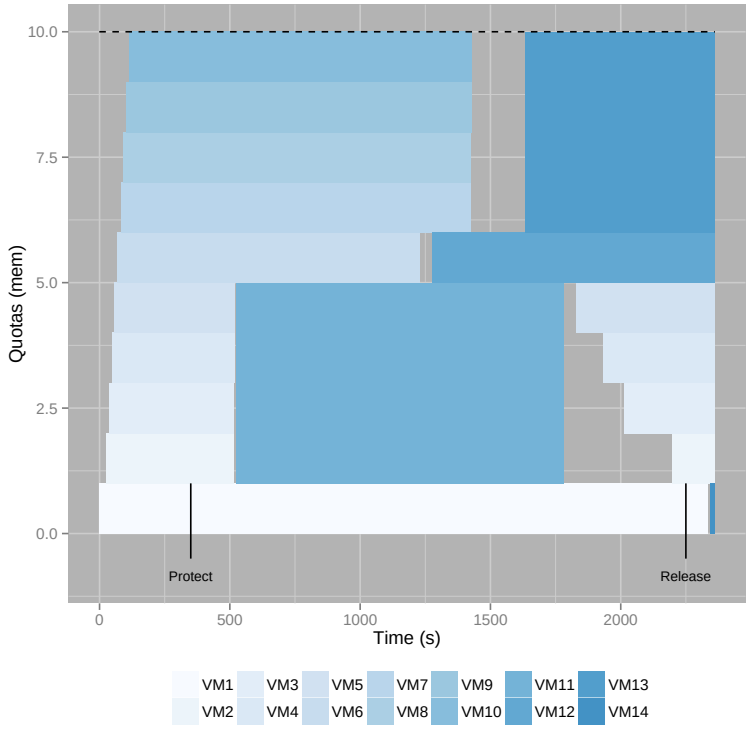


Figure 4.8: Pinning VM instances with CloudGC.

Among improvements, we are considering the integration of a VM consolidation phase in the recycling process of CloudGC. Given the overhead of VM consolidation, we plan to trigger such a phase only when CloudGC requires to compact the resources to ease the provisioning of larger VM instances. In such a case, the VM consolidation phase will aim at grouping *i)* pinned VM instances

on a subset of compute nodes and *ii)* the resources made available by CloudGC on a single node. Among the possible solutions to implement this phase, we are considering the integration of CloudGC with the Watcher service[6], which has been recently released by OpenStack.

4.4.2 CloudGC Performance Analysis

Regarding the delay introduced by the recycling process of CloudGC, the previous sections profiled the phases of CloudGC to identify how it performs depending on the different situations considered in the scenario (detailed in Section 4.1). Figure 4.9 therefore reports on the completion times achieved by CloudGC to provision new or recycled VM instances in the Cloud infrastructure.

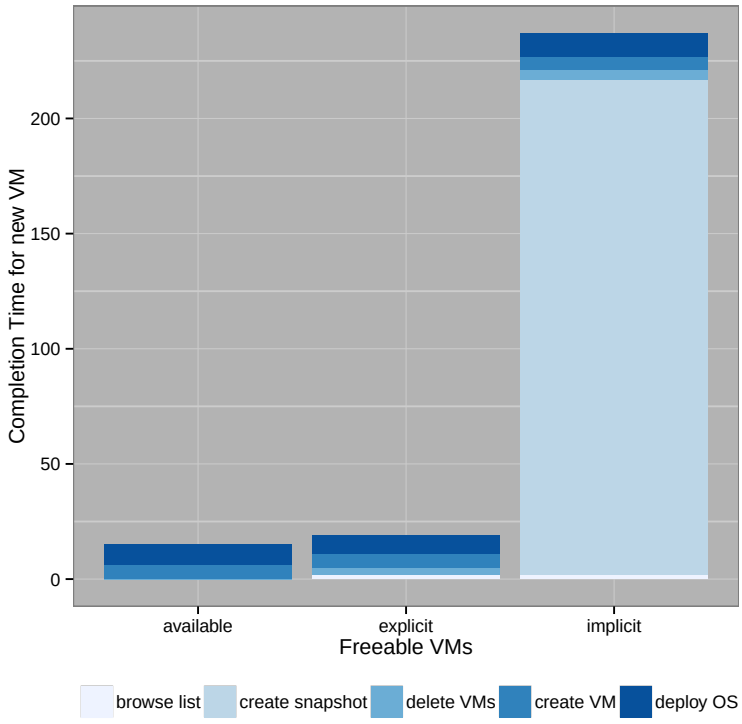


Figure 4.9: Recycling delays introduced by CloudGC.

As long as enough resources are available in the Cloud infrastructure, one can observe that the recycling process of CloudGC does not include any processing overhead for the system, thus performing equally to a standard configuration of

[6] `https://wiki.openstack.org/wiki/Watcher`

OpenStack. When CloudGC recycles explicitly idle VM instances, one can assess that the processing overhead of CloudGC is rather low limited compared to a standard provisioning process, adding only 5 seconds to recycle an idle VM that has been explicitly suspended (cf. Table 4.2).

The biggest processing penalty introduced by CloudGC correspond to the recycling of implicitly idle VM instances that are in a running state. In this specific case, CloudGC requires to take a snapshot of the VM instance right before releasing the associated resources, which impose to wait for the image to be safely persisted on the storage device before completing the provisioning process, and thus explaining the 215 seconds taken by Glance to complete this phase.

Table 4.2: Processing overhead per phase.

operation	available	explicit	implicit
browse list	-	2 *sec*	2 *sec*
create snapshot	-	-	215 *sec*
delete instance	-	3 *sec*	3 *sec*
create instance	6 *sec*	6 *sec*	6 *sec*
deploy OS	9 *sec*	9 *sec*	9 *sec*
total	15 *sec*	20 *sec*	235 *sec*

Regarding memory consumption, Figure 4.10 compares the memory consumption of OpenStack with and without CloudGC in Cloud controller node. On average, the difference between the two curves represents an overhead of 50 MB for the Cloud controller node, on which CloudGC is deployed with the other infrastructure services. During the provisioning phases, which are reflected as peaks in Figure 4.10, one can observe that the memory overhead of CloudGC may reach up to 100 MB due to the additional activities performed as part of the recycling process.

Regarding the storage consumption, the storage capacity of Glance is impacted by CloudGC as it uses this service to store the snapshots of recycled VMs. The storage overhead of CloudGC therefore corresponds to the number of currently recycled VMs times the size of a VM, which highly depends on the activity of the Cloud. For example, Figure 4.6 provides an estimation of the volume of VMs recycled by CloudGC and thus subsequent snapshot images it has to store. CloudGC therefore trades CPU and memory resources against storage resources,

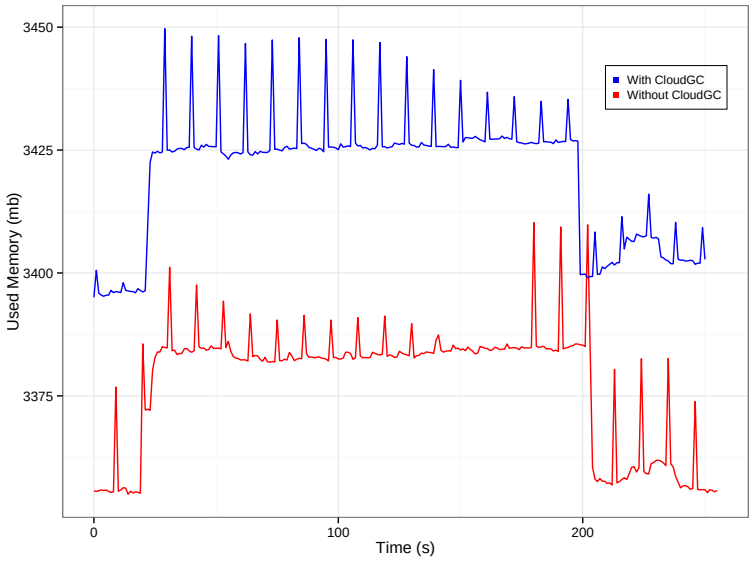


Figure 4.10: Memory overhead introduced by CloudGC.

but we assume that the resource limitations of a Cloud are stronger when it comes to CPU and memory resources.

With regards to current limitations, we are therefore exploring solutions to reduce the impact of on-demand snapshotting, which is the major bottleneck of CloudGC when recycling implicit idle VM instances. In particular, we are considering the support for incremental snapshots of *idle* VM instances to reduce both the processing and the network overhead imposed by the snapshot operations. By integrating such an incremental snapshot mechanism, CloudGC aims at completing the snapshot associated to an idle VM instance whenever the monitoring component detects that its internal state has changed.

4.4.3 Orchestrating Periodic Usages

Thanks to CloudGC, a single physical infrastructure can be shared by several groups of VM instances that do not operate continuously. OpenStack can therefore periodically and automatically switch between VM instances along periods of variable durations in order to keep delivering the requested services, according to user requirements. For example, a Cloud infrastructure can host a group of services during office hours, then switch to another group of VMs during night before moving to a third profile along week-ends. While supporting this kind of scenario requires carefully handcrafted scripts to orchestrate the groups of VMs in

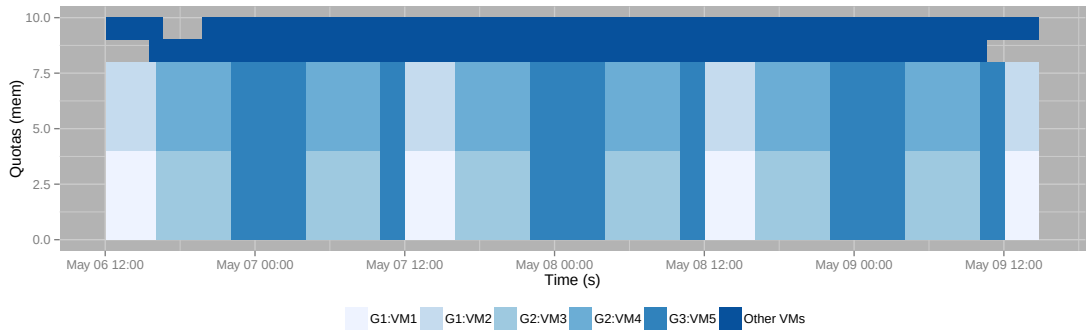


Figure 4.11: Supporting periodic VM instances in OpenStack with CloudGC.

OpenStack, CloudGC delivers this support natively, by exploiting the features we detailed in this chapter.

For example, Figure 4.11 reveals a 4-day experiment we run in the OpenStack infrastructure running CloudGC. In this experiment, we provision 5 VMs, which are operating in 3 groups, next to other standard VM instances, which can be provisioned and used more randomly. The group 1 includes two small VM instances, which are used from noon every day. The group 2 contains two other small VM instances, which need to be available twice a day at 4:00 and 16:00, respectively. Finally, the group 3 has a single medium VM instance, which is usually active from 22:00 to 10:00.

From Figure 4.11, one can observe that, beyond the traditional usage of a Cloud infrastructure, CloudGC succeeds to schedule periodic VM instances according to their respective requirements. In order to guarantee the availability of periodic VM instances on time, we also benefit from the timer of CloudGC (cf. Section 4.3.4) to ensure that the VM instances are provisioned 5 minutes before the expected time. Furthermore, the use of the timer turns out to be also useful for autonomous VM instances, which do not need to be requested by an external user and control their own activity.

The perspectives regarding this support for periodic VM instances refers to the automatic mining of VM activity patterns in order to configure the timer of CloudGC automatically. By adding such a capability, we believe that CloudGC can evolve to provide a new building block of a Cloud infrastructure saving energy by turning off the compute nodes hosting idle VM instances during periods of

inactivity (*e.g.*, nights, week-ends, holidays). Furthermore, the combination of CloudGC with an elasticity service would enable compute nodes to be waken up automatically by the timer or by requesting one of the recycled service.

4.5 Summary of CloudGC

While Cloud computing conveys the image of unlimited virtual resources that can be allocated on-demand, the state-of-practice shows that the Cloud infrastructure may limit the flexibility. Recently, beyond major Cloud providers (*e.g.*, Google) who manage large data centers, private Clouds are also being widely deployed. The OSS solutions like OpenStack are providing the necessary services to deploy an IaaS atop of clusters with a small size. Compared to public Cloud, the private Clouds may quickly reach its limits. Due to the ossified resource allocation mechanism, the OpenStack Cloud may suffer from infrastructure limitations when satisfying the user requests. While Cloud elasticity is now considered as the *de facto* solution to scale the Cloud, we advocate that Cloud elasticity should be carefully triggered as it may induce some non-negligible operational costs with budget and carbon emission implications.

In this chapter, we therefore propose to identify potential resource leaks in a Cloud infrastructure to develop a non-elasticity approach. Based on the experiments, one can find that *Idle* VMs—*i.e.*, instances that are either suspended or inactive for a while—occupy lots of idle resources which cannot be re-allocated for others, thus causing resource leaks. To address the problem, our new approach automatically recycles *idle* VM instances in order to provision new VMs upon requests. The middleware service, named CloudGC, detects periodically such *idle* VMs and recycles them to free the resources, which are necessary to satisfy new VM provisioning request. At the moment, CloudGC is integrated as an OpenStack service, thus not only providing a seamless support for typical services (such as recycling *idle* VMs and recovering them), but also offering the periodic deployment of VMs depending on user requirements. The evaluations demonstrate that CloudGC can push the limits of the Cloud infrastructure beyond the standards to mitigate the impact of resource leaks on Cloud flexibility, and the recycling overhead introduced by CloudGC also remains in an acceptable range.

Short-term perspectives on this work include the integration of additional services of OpenStack, such as Neutron and Cinder, in order to provide a full support for more complex VM configurations. In order to further improve the performance of the solution, the extensions of CloudGC—that should include *(i)* the automatic consolidation of VM in order to optimize the utilization of Cloud resources, *(ii)* the support for incremental VM snapshots in order to reduce the latency of the recycling process, and *(iii)* the mining of VM activities in order to identify and anticipate recurring activities of VMs in a Cloud—also begin to be considered.

4.6 Synthesis

Thanks to the resource optimization (CloudGC) in OpenStackCloud, Cloud infrastructure limitation often will not disturb users by insufficient capacities, thus guaranteeing and improving the flexibility of Cloud computing. CloudGC aims to avoid the resource waste in the provisioned environments, while ensuring a reasonable resource utilization for the Cloud infrastructure. This therefore complements the other researches concerning elasticity to make sure they will be invoked at necessary moment, thereby avoiding the over-provisioning problem—*i.e.*, CloudGC is able to improve their effect. In particular, the resource optimization strengthens the spontaneity of elasticity—*i.e.*, in the case without any of resource waste—when the provisioned resources (current Cloud infrastructure) still cannot fit the user requirements, the elasticity is proven to become necessary at that moment.

However, focusing only on the resource waste at the infrastructure-level is far from enough to guarantee the QoS in Cloud computing. As introduced in Chapter 1, many Public Cloud providers recently begin to provide Hadoop-related services (*e.g.*, Amazon EMR) to facilitate the business and developments in Big Data domain. OpenStack also has the same plan to launch a sub-project *Sahara*, which helps Cloud users to quickly deploy and easily manage an Hadoop cluster atop of OpenStack. In this case, the Hadoop-related researches become more and more interesting and important to Cloud computing. Therefore, my thesis covers also the optimizations applicable on Hadoop clusters. By experimenting with some well-known Hadoop benchmarks, we reveal several problems in memory

utilization, which are associated to YARN the new component introduced into Hadoop since an overhaul in 2012. We find that these problems are caused by the static configurations and YARN itself. To solve them, a self-balancing approach is proposed to adjust the YARN configurations at runtime. The problems and the approach will be further illustrated and evaluated in next chapter.

Chapter 5

Resource Optimization in Hadoop

According to the rise of business and development in Big Data domain, many Cloud providers begin to support Hadoop-related services, such as Amazon EMR, etc. Even OpenStack, an open-source project which aims to accelerate and facilitate the development and management of a private Cloud, has launched a sub project, named *Sahara*, to help end-users to easily deploy and manage Hadoop cluster atop of an OpenStack Cloud. In this case, the Hadoop performance also affects the QoS in Cloud computing.

As a well-known distributed data-processing environment, Hadoop is widely used in various Big Data business to process ad-hoc request or concurrent workloads. For the ad-hoc request, especially the one processing a huge data set, users are able to prepare a customized Hadoop cluster to maximize the service performance. In particular, many researchers introduced machine learning technologies to automatically achieve the customization of Hadoop cluster for ad-hoc requests. Beyond ad-hoc requests, Hadoop cluster also processes concurrent workloads. However, the dynamics of concurrent workloads often cause distress for the performance of Hadoop cluster, particularly in the case of time-varying workloads. The distribution of concurrent jobs, even the changes of job sizes, are prone to result in the resource leaks of Hadoop cluster, due to its unchangeable configurations which cannot be adapted to the dynamics at runtime, thus causing the degradation of service performance. Furthermore, Hadoop cluster also adopts a master-slave architecture, which allows the cluster administrators to easily scale an Hadoop infrastructure (VMs in Cloud computing). This attracts lots of attention from elasticity researches for Hadoop resource management to adapt a cluster

according to the dynamics of concurrent workloads. My research therefore focuses on the resource optimization of Hadoop cluster.

As introduced above, the resource optimization in Hadoop cluster can be divided into two parts:

- **Resource Management** : Hadoop is a complex distributed data-processing environment, which contains hundreds of configuration parameters. Due to the complexity of configurations, configuring Hadoop cluster requires lots of experience as the service performance is often affected by misconfiguration (which is prone to cause resource leaks, thus degrading system performance). For the time-varying concurrent workloads, this problem become more serious. Beyond the issue in Hadoop configurations, the dynamics of concurrent workloads also raise several challenges to Hadoop cluster : *i*) the Hadoop static configurations should be able to be dynamically tuned at runtime; *i*) the modifications on configuration should adapt the Hadoop cluster to the dynamics of workloads; *i*) the solution should be able to detect the performance degradations of an Hadoop cluster to avoid unnecessary actions, which do not improve the performance but in turn degrade it. In this case, I therefore focus on the dynamic configuration of Hadoop cluster to eliminate the resource leaks caused by misconfiguration, thereby providing the optimal Hadoop performance for concurrent workloads based on the current infrastructure.

- **Rapid deployment** : As a distributed data-processing environment, Hadoop performance highly depends on the infrastructure where it deploys. This bothers many researchers when evaluating their works. In particular, to compare the performance of Hadoop-related approaches, researchers must reproduce the same Hadoop cluster for all approaches. As introduced before, deploying Hadoop cluster is a time-consuming and complex work. In this case, I advocate a new tool to simply and accelerate the deployment of a complete Hadoop cluster, which is based on Docker technologies. Furthermore, the elasticity can differentiate from scalability because it does not only consider how to scale the infrastructure but also consists of many other issues, such as spontaneity and timeliness (*i.e.*, rapid deployment). Therefore, this tool should be

also interesting for those focusing on other Hadoop research, such as elasticity and self-adaptation etc.

The two Hadoop-based researches will be illustrated in the following sections, respectively.

5.1 Resource Management in Hadoop

As introduced in Chapter 3, an Hadoop cluster is a typical distributed system with a master-slave architecture, thus its performance can be improved by classical scalability and elasticity approaches. However, these approaches have a common loss—*i.e.*, they all selectively disregard the effort of improving resource usage in performance optimization. Even though these approaches can guarantee the Hadoop performance, but also introduces a growth cost. In this chapter, we will propose a new approach to improve Hadoop performance, based on related works, which do not only contain the classical approaches, but also consists of many others. And, we believe that it will become an important complement to elasticity and many other approaches.

Along the years, Hadoop has emerged as the *de facto* standard for Big Data processing. One of Hadoop popular processing framework, named the MapReduce paradigm, has been applied to a large diversity of applications and workloads, including distributed sorting, log analysis, document clustering, machine learning, etc. In this context, the performance issues of Hadoop has attracted more and more attention in research and industrial communities.

The performance and the resource consumption of Hadoop jobs do not only depend on the characteristics of applications and workloads, but also on an appropriately configured Hadoop environment. The Hadoop environment is controlled by two parts: infrastructure configuration (*e.g.*, the number of nodes in a cluster) and Hadoop platform parameters. Hadoop contains hundreds of parameters, which can be generally divided into two categories: *job-oriented* and *system-oriented*.

- *job-oriented* parameters are responsible to regularize the submitted jobs. These parameters ensure that the executing jobs comply with the standard of Hadoop, thus avoiding the failures of job processing.

- *system-oriented* parameters support the execution of an Hadoop cluster. They control system-level configurations, such as Hadoop component connections, resource allocation settings, job parallelism, etc.

Next to the infrastructure-level configuration, the Hadoop performance is significantly affected by Hadoop parameter (both job- and system-level parameters) settings. Optimizing the job-level parameters to accelerate the execution of Hadoop jobs has been a subject to lots of research work [14, 28, 33, 36, 37, 38]. These approaches tend to either optimize specific job categories and workload patterns, or use machine learning techniques to increase their application. Nevertheless, because of the diversity of Hadoop jobs and the large number of job-level parameters, machine learning requires a large training set for building an accurate internal model. This obviously reduces the effectiveness of machine learning.

Beyond job-level configuration, Hadoop also includes a large set of system-level parameters. In particular, YARN (*Yet Another Resource Negotiator*), the resource manager introduced in the new generation of Hadoop (version 2.0) defines a number of parameters that control how the applications (*e.g.*, MapReduce jobs) are scheduled in a cluster, which influence jobs performance. YARN is a general scheduler allowing to run various distributed application beyond MapReduce (*e.g.*, Spark, Flink, Tez, GraphLab), but the focus in this research is primarily on MapReduce jobs.

Without considering the performance issues of job-level configurations, due to the diversity of high-level applications, the static configuration of Hadoop system and its corresponding researches are still not enough to guarantee the optimal Hadoop performance. On the one hand, in the case of this research, the diversity of MapReduce applications and workloads suggests that a simple, *one-size-fits-all* application-oblivious configuration will not be broadly effective—*i.e.*, one Hadoop configuration that works well for one MapReduce application/workflow combination might not work for another [52]. On the other hand, YARN configuration is static and as such, it cannot reflect any changes in workloads dynamics. The only possibility is to do a *best-effort* based on either experience or a static profiling in the case the jobs and workloads are known *a priori*. However, (1) this might not be always possible to adjust system-level parameters at runtime, (2) it requires

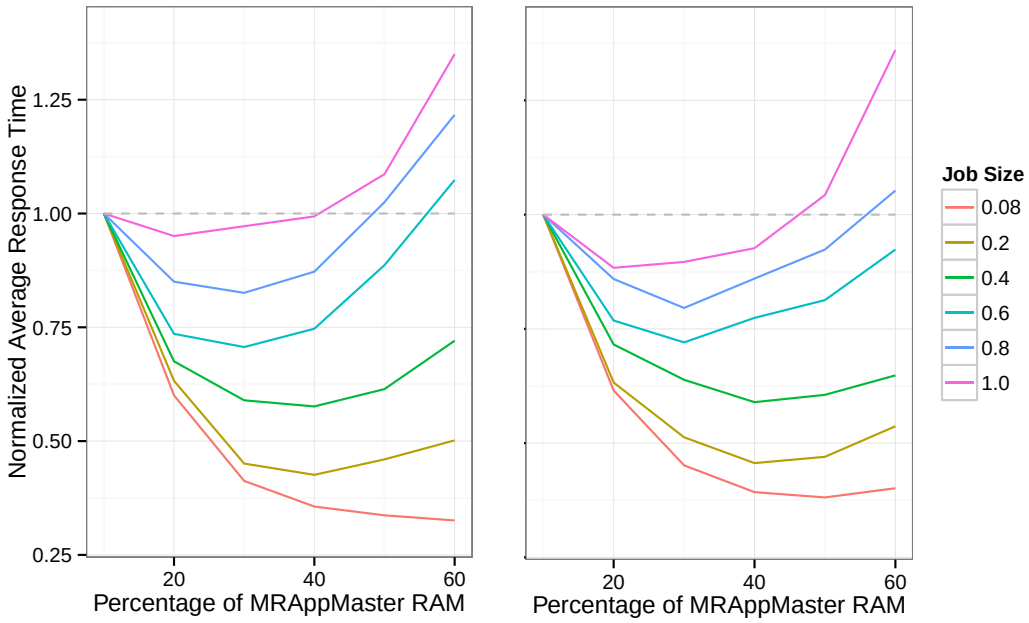
additional work to analyze the submitted jobs and workloads, and (3) any unpredictable workload changes (*e.g.*, a load peak due to node failures) will cause performance degradation.

Among YARN parameters, the MARP (*Maximum Application master Resource in Percent* : yarn.scheduler.capacity.maximum-am-resource-percent) property directly affects the level of MapReduce job parallelism and associated throughput. This property balances the number of concurrently executing MapReduce jobs versus the number of the corresponding map/reduce tasks. An inappropriate MARP configuration will therefore either reduce the number of jobs running in parallel resulting in idle jobs, or reduce the number of map/reduce tasks and thus delay the completion of jobs. However, finding an appropriate MARP value is far from trivial. The balance between job parallelism and throughput can significantly impact Hadoop performance. Meanwhile, according to the increase of MARP value, Hadoop performance actually expose a non-monotone behavior which will be shown in Section 5.1.1. Moreover, thanks to YARN privilege commands, Hadoop cluster can reload the MARP parameter at runtime. This provides the possibility to tune MARP towards a *best-effort* value for the running jobs and workloads without bothering the whole Hadoop cluster.

In this research, we therefore focus on dynamic MARP configuration. First, we identify the relationship between the MARP parameter and the performance of variety of MapReduce applications and workloads using established benchmarks. Second, based on the analysis, we implement a self-configuration heuristics as a feedback control loop that continuously adjusts the MARP parameter at runtime.

The evaluation demonstrates that the approach systematically achieves better performance than static configuration approaches. Concretely, one can outperform the default Hadoop configuration by up to 40% and up to 13% for the *best-effort* statically profiled configurations, yet without any need for prior knowledge of the application or the workload shape, nor any need for any learning phase. The main contributions of the research are:

(1) an analysis of the effects of the MARP parameter on the MapReduce job parallelism and throughput, and



(a) 100 requests/peak (b) 50 requests/peak

Figure 5.1: Relationship between the job size, the percentage of RAM for `MRAppMasters`, and the average job response time.

(2) a feedback control loop that self-balances the MapReduce job parallelism and throughput.

In the next section, we will introduce the architecture of YARN where this research relies on.

5.1.1 Limitations of Static Configurations

To understand the limitation of static configuration, we first study how the MapReduce job size (*i.e.*, the number of tasks to be processed) and the MARP parameter affect the overall completion time of Hadoop jobs. All experiments were performed using an Hadoop cluster made of 11 physical hosts[1] (1 control node and 10 compute nodes) deployed on the Grid5000 infrastructure [2]. The Hadoop version is 2.6.1.

PI estimation is a default MapReduce performance benchmark packaged in Hadoop. It allows users to specify the input workloads by adjusting MapReduce job size. We repeat the experiment to cover all the combinations between MARP value and MapReduce job size. Figure 5.1 exposes the mean completion time of

[1] 2 Intel Xeon L5420 CPUs with 4 cores, 15GB RAM, 298GB HDD
[2] https://www.grid5000.fr

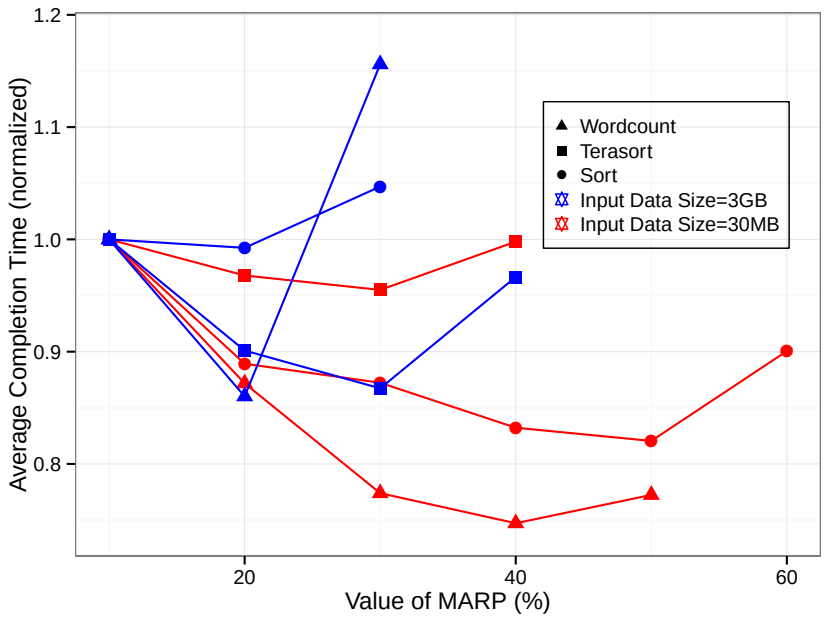


Figure 5.2: Effects of different MARP configurations, job type and job size on mean completion time of 100 jobs.

two PI ESTIMATION load peaks—*i.e.*, 50 jobs/peak and 100 jobs/peak. One can observe that (1) the default MARP value cannot provide the optimal performance, and (2) the Hadoop performance shows non-monotone behavior according to the increase of MARP value. To ensure the figure visibility, all the completion times are normalized according to the absolute completion time of the vanilla Hadoop configuration—*i.e.*, MARP = 0.1.

Figure 5.2 reports on the completion time of the three applications provided by the HIBENCH benchmark suite [31]: Wordcount, Terasort, and Sort. For each of the input workloads—*i.e.*, 30MB and 3GB—one can observe the impact of the MARP parameter on the mean completion time of 100 jobs.

As expected, the vanilla configuration does not provide the best performance for any of the workloads. Furthermore, one can observe that the best performance is not achieved by a single value of MARP, but rather tends to depend on the size of the job. In particular, increasing the value of MARP—thus allocating more resources to the MRAppMaster containers—tends to benefit the smaller Hadoop jobs, while large jobs complete faster when more resources is dedicated to the YarnChild containers.

Next, we stress the Hadoop cluster by running a different number of jobs in

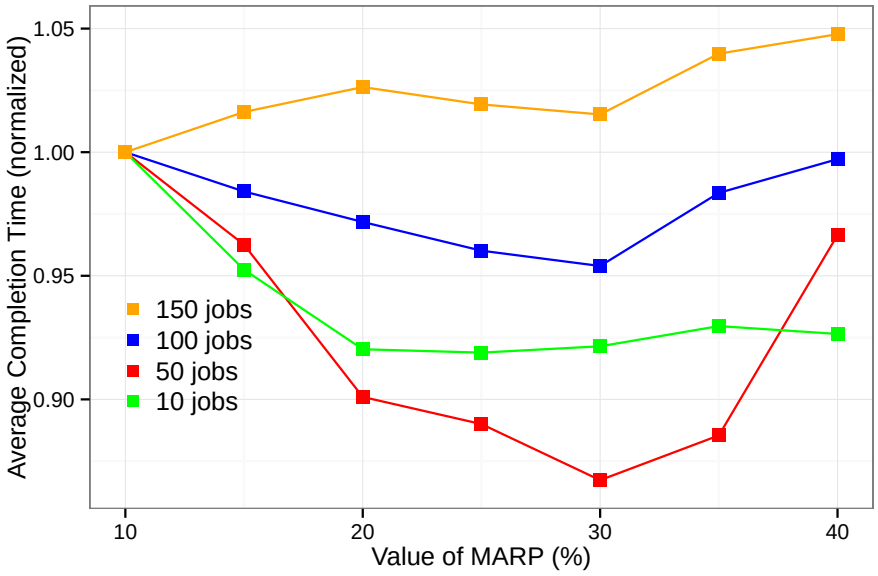


Figure 5.3: Effects of different MARP configurations and load peak stress on mean completion time.

parallel in order to observe the impact of a load peak on the job mean completion time. Figure 5.3 shows the performance when running Terasort with 3GB workload under various stress conditions. Compared to Figure 5.2, one can observe that by increasing the number of concurrently running jobs, the optimal value of MARP differs from the previous experiment. In this case, we can find that, the size of job is not the only characteristic which affects the optimal MARP value, but the size of workloads also does. Therefore, while a MapReduce job can be profiled for a *best-effort* MARP configuration in a specific Hadoop cluster, any unpredictable changes in the workload dynamics will lead to a performance degradation.

Finally, we consider heterogeneous workloads. Concretely, we use SWIM (*Statistical Workload Injector for Mapreduce*) [15] to generate 4 realistic MapReduce workload. SWIM containing several large workloads (thousands of jobs), with complex data, arrival, and computation patterns that were synthesized from historical traces from Facebook 600-nodes Hadoop cluster. The proportion of job sizes in each input workloads has been scaled down to fit the cluster size using a Zipfian distribution [3]. For example, Figure 5.4 shows the job distribution synthesized from SWIM that was used in the experiment as the first workload (W1).

As previously observed for homogeneous workloads, Figure 5.5 demonstrates

[3]http://xlinux.nist.gov/dads/HTML/zipfian.html

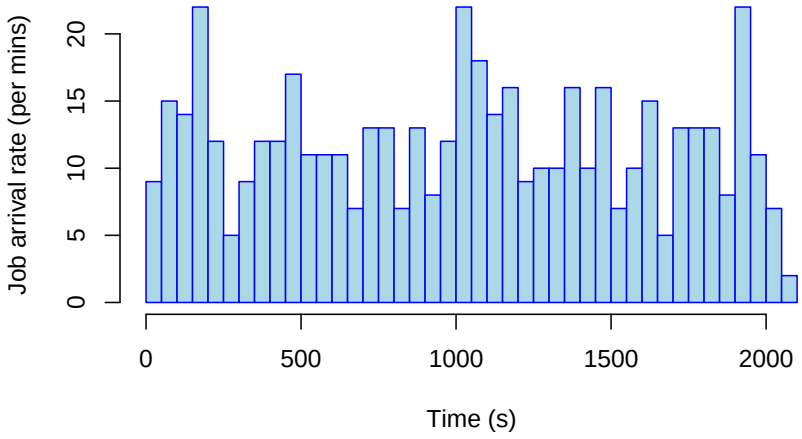


Figure 5.4: A job distributions generated by SWIM used for W1.

that a single MARP value cannot fit all the workloads and the best configuration can only be set by having a deep understanding of the Hadoop jobs and their dynamics.

Synthesis. These preliminary experiments illustrate that the MARP configuration has a clear influence on Hadoop performance. They show that the default value is not optimal for almost all the considered cases. While one can profile the different applications to identify the *best-effort* configuration we have shown that any unforeseen change in the workload dynamics can degrade the overall performance. To leverage the required expertise, we therefore advocate for a self-adaptive approach that continuously adjusts the MARP configuration based on the current state of the Hadoop cluster.

5.1.2 Memory Consumption of Hadoop

As shown in the previous section, the static configuration of MARP can cause several types of performance issues. Since containers currently consider only memory, in this section we focus on memory consumption and analyze the causes of the performance bottlenecks.

In an Hadoop cluster, the memory can be divided into 3 parts: M_{system}, M_{Jobs}, and M_{idle}. M_{system} is the memory consumed by the system components—*i.e.*, Re-

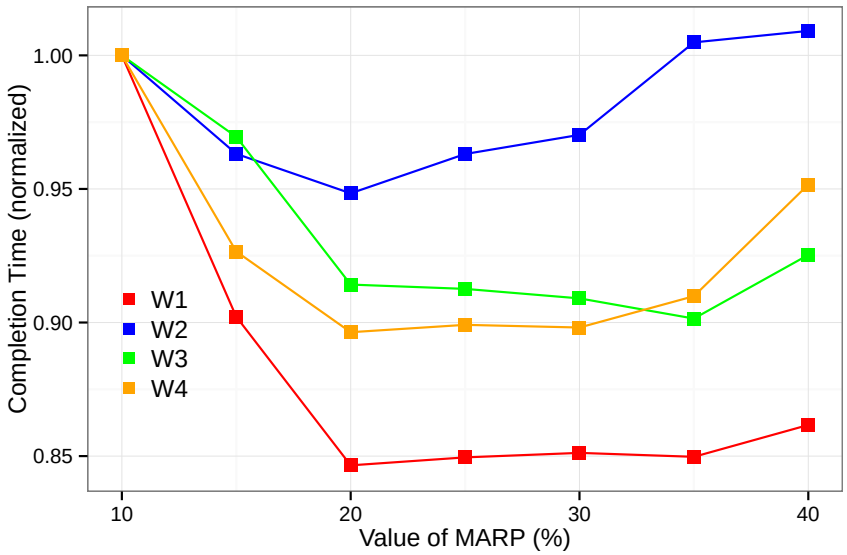


Figure 5.5: Effects of different MARP configurations and different SWIM generated workloads on overall completion time.

sourceManager and NodeManager in YARN and NameNode, DataNode in HDFS. The memory consumption of this part is almost constant. The other three parts represents the memory held by NodeManager(s) as a result of processing MapReduce jobs. M_{Jobs} contains two parts: M_{AM} and M_{YC}, they are all the memory occupied by MapReduce jobs, but consumed by MRAppMaster and YarnChild, respectively. The relationship between M_{system}, M_{Jobs}, M_{AM}, M_{YC}, and M_{idle} with the overall memory of a Hadoop cluster, $M_{compute}$ can be expressed as follows:

$$M_{compute} = M_{system} + \underbrace{\underbrace{\sum_{1}^{n} M^{n}_{AppMaster}}_{M_{AM}} + \underbrace{\sum_{1}^{n}\sum_{1}^{m} M^{nm}_{YarnChild}}_{M_{YC}}}_{M_{Jobs}} + M_{Idle} \tag{5.1}$$

Upon starting an Hadoop cluster, $M_{compute}$ is fixed (unless new computing nodes are enlisted or existing discharged from the cluster).

$M_{AM} = \sum_{1}^{n} M^{n}_{AppMaster}$ is the memory allocated to all the MRAppMaster containers across all compute nodes. This is controlled by the MARP configuration—*i.e.*, $M^{*}_{AM} = M_{compute} \times \text{MARP}$. During the processing of jobs, $M_{AM} \leqslant M^{*}_{AM}$.

$M_{YC} = \sum_{1}^{n}\sum_{1}^{m} M^{nm}_{YarnChild}$ is the memory used by all the YarnChilds to process map and reduce tasks across all the concurrently running jobs on all the computing nodes. This part directly impacts the job processing rate. A

larger M_{YC} means that the more map or reduce tasks can be launched in parallel and the faster ongoing jobs are completed.

M_{idle} is the unused memory across all the computing nodes. High M_{idle} value together with pending jobs is a symptom of a waste of resources. It typically means that the system performance have a room for improvement.

5.1.2.1 Loss of Jobs Parallelism

The maximum number of concurrently running jobs, N_{max}, in an Hadoop cluster is

$$N_{max} = \frac{M^*_{AM}}{M_{container}} \tag{5.2}$$

where $M_{container}$ is the NodeManager container size (by default it is 1GB). The smaller the MARP value is, the smaller N_{max} will be and the less jobs will be able to run in parallel.

In the case that the number of running jobs equals to N_{max}, all available application master containers are exhausted and ResourceManager cannot schedule any more jobs into Hadoop cluster for processing. The new submitted jobs therefore become idle, and wait for the permissions in a queue. In this case, Equation (5.1) can be rewritten as follows:

$$M_{compute} = M^*_{AM} + M_{YC} + M_{idle} \tag{5.3}$$

Where M_{idle} will emerge with a low N_{max} (*i.e.*, low MARP value). When the number of running jobs reaches N_{max}, $M_{AM} = M^*_{AM}$ and no more pending jobs can be run even though Hadoop cluster has M_{idle} (*i.e.*, $M^*_{AM} + M_{YC} < M_{compute}$). Therefore, according to Equation (5.3), one can observe that the lower $M^*_{AM} + M_{YC}$ is, the higher M_{idle} is, indicating a memory / container waste that in turn degrades performance. One call this situation the *Loss of Jobs Parallelism* (LoJP). Figure 5.6 illustrates such a situation. An Hadoop cluster with 8 containers has the MARP value set too low allowing only one job to be executed concurrently. Any pending jobs will have to wait until the current job has finished despite that there are unused containers. In this case, the waiting time of each pending job increases significantly.

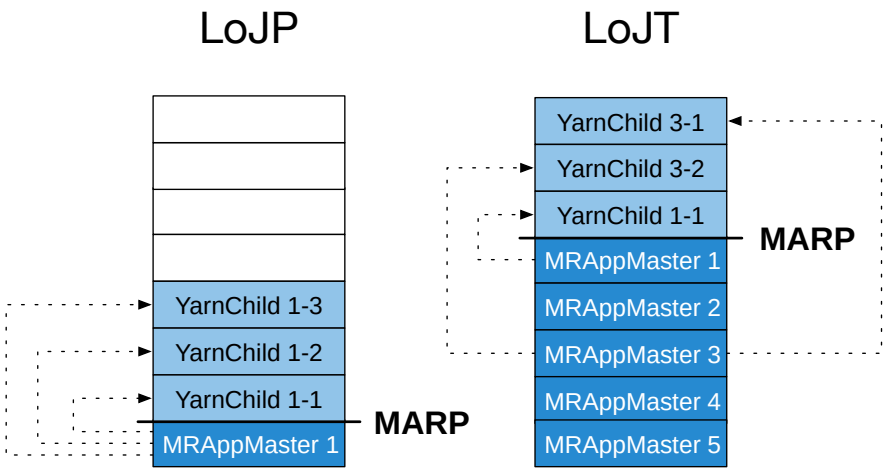


Figure 5.6: LoJP and LoJT in Hadoop.

5.1.2.2 Loss of Job Throughput

As shown in the previous section, small N_{max} limits the jobs parallelism within an Hadoop cluster. However, large N_{max} may also impact the job performance.

By increasing N_{max} (or M^*_{AM}) in order to absorb M_{idle}, Equation (5.1) can be rewritten to:

$$M_{compute} = M_{AM} + M_{YC} \tag{5.4}$$

In this case, the memory utilization is not probably limited by MARP. Once a new job is submitted, it will be directly scheduled into Hadoop cluster until M_{idle} is exhausted. Compared to LoJP, M_{idle} is eliminated as far as possible. The waiting time of pending jobs theoretically should be reduced, indicating a short completion time that in turn improve performance.

However, when the Hadoop cluster processes a large number of concurrent jobs, M_{AM} becomes a major part of $M_{compute}$ and thus it limits M_{YC}. MRAppMaster is a job-level controller and it does not participate in any map or reduce task processing. Therefore, a limited M_{YC} decreases significantly the processing throughput of an Hadoop cluster. This symptom is identified as a *Loss of Job Throughput* (LoJT) and is also illustrated in Figure 5.6. In this case, we have set the MARP too high, which allows many jobs to run in parallel, yet the actual processing capacity is limited by the low number of available container for running YarnChild. In this case, the waiting time of each pending job will decrease slightly, but the processing time of each running job will increase significantly. Furthermore, for

the new submitted jobs, they might endure a long processing time as well as a long waiting time caused by previous jobs. In a saturated Hadoop cluster, LoJT will not improve performance, but conversely degrade it.

5.1.2.3 Large Drops of Memory Utilization

Depending on the size of the jobs and the memory used in YarnChild containers, the dynamic allocation of resources can result in large drops of memory utilization (cf. Fig. 5.7). This is especially true when the tasks are rather fast to complete.

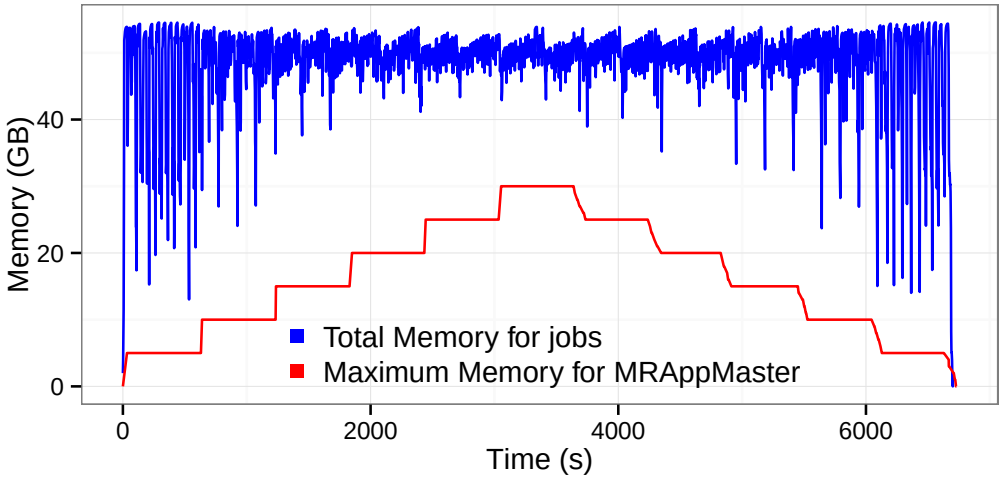


Figure 5.7: Amplitude of memory drops depending on the MARP value.

In this scenario, the deployed Hadoop job consists of 20 map and 1 reduce tasks deployed in a cluster provisioned with $50GB$ of memory that we stress with a continuous flow of requests to ensure that the number of running jobs is close to N_{max} (cf. Equation 5.2). By increasing the MARP value every 10 minutes, one can follow the impact of jobs parallelism on the memory utilization of the cluster. In particular, one can observe in Figure 5.7 that the lower MARP value, the larger and more frequent memory drops—even for large jobs, which are expected to benefit from low MARP value (cf. Section 5.1.2.1).

These memory drops are caused by the different lifecycle of *containers*, and basically appear at the end of concurrently running jobs. When a job comes to the end, all its corresponding M_{YC} will be quickly released. However, its MRAppMaster is still running to organize data, and to report results to users. Due to the running MRAppMaster, idle jobs cannot get the permission to access mem-

ory for processing. In this case, if other concurrently running jobs do not have enough unscheduled map or reduce tasks to consume these M_{idle} (released M_{YC}), the memory utilization will drop. A higher MARP value means more concurrently running jobs, which probably have more unscheduled map or reduce tasks to avoid the memory drops, and vice versa.

To clearly display the process of **Large Drops of Memory Utilization**, Figure 5.8 reveals the changes of memory utilization during this process. In this scenario, MARP permits only 1 MapReduce job processing in parallel.

1) At time T0, Job1 is processing in Hadoop cluster. It has 1 MRAppMaster and m YarnChild which make memory utilization high.

2) All m YarnChild have finished at T1. However, MRAppMaster of Job1 is still active for collecting results, recording status and reporting to users. Because of the MARP limitation and active MRAppMaster of Job1, another pending jobs cannot be scheduled into Hadoop cluster even though memory utilization becomes low.

3) As introduced before, MRAppMaster is the first *container* of application launched by ResourceManager. In this context, after the end of Job1, Job2 gets its MRAppMaster to begin the preparation of processing at T2, such as splitting datasets, negotiating resources, and launching YarnChild. During this period, only MRAppMaster of Job2 is running in Hadoop cluster, thus the memory utilization is still low.

4) When MRAppMaster has finished the preparation, the launched YarnChild also began to process its corresponding tasks. At the moment T3, Job2 is processing in the Hadoop cluster. The memory utilization therefore becomes high again.

In this scenario, one can obviously find that, between T1 and T2, the memory utilization of Hadoop cluster remains in a low level. Therefore, a **Large Drop of Memory Utilization** appears.

The memory drops cause temporarily high M_{idle}, and therefore reduce the average memory utilization—*i.e.*, this phenomenon also contributes to performance

$$\begin{aligned}
&\text{T0 - Job1 : } M^1_{AppMaster} + \textstyle\sum_1^m M^{1:m}_{YarnChild} \quad (MU : High) \\
&\text{T1 - Job1 : } M^1_{AppMaster} \quad (MU : Low) \\
&\qquad\qquad\qquad\qquad \Longleftrightarrow \text{ One Large Drop of Memory Utilization} \\
&\text{T2 - Job2 : } M^2_{AppMaster} \quad (MU : Low) \\
&\text{T3 - Job1 : } M^2_{AppMaster} + \textstyle\sum_1^m M^{2:m}_{YarnChild} \quad (MU : High)
\end{aligned}$$

Figure 5.8: The process of 1 Large Drop.

degradations. Moreover, the frequent and large memory drops can also lead the users to misjudge the state of the Hadoop cluster.

5.1.3 Self-balancing Proposition

Based on the previous section, we propose a self-adaptive approach for dynamically adjusting the MARP configuration based on the current state of the cluster.

5.1.3.1 Maximizing Jobs Parallelism

The symptom of LoJP—*i.e.*, small N_{max}, large M_{idle} leading to decrease the memory utilization— can be detected from the ResourceManager component and fixed by increasing the MARP parameter. However, it should not consequently cause LoJT (cf. Section 5.1.2.2). We therefore propose a greedy algorithm to gradually increase the MARP parameter (cf. Algorithm 5). It is a simple heuristics that periodically increments MARP by a floating step (inc) until a given threshold (T_{LoJP}) is reached—*i.e.*, the overall memory consumption $M_U = M_{AM} + M_{YC}$ falls below the threshold $M_U < T_{LoJP}$. Both, the current M_U and MARP values can be observed from ResourceManager. Once the increment becomes effective, ResourceManager will continue to schedule any pending jobs until the N_{max} limit is reached. A short delay between the increment steps ($delay$) is therefore required to let the cluster settle and observe the effects of the increment.

5.1.3.2 Maximizing the Job Throughput

The LoJT symptom is more difficult to detect since, at the first glance, the Hadoop cluster appears to fully utilize its resource. However, as shown in (5.4), this situation can be also a result of the cluster saturation with too many jobs run-

Algorithm 5 Fixing LoJP by incrementing MARP.

```
1: procedure LoJP(T_LoJP, inc, delay)
2:     M_U ← current memory utilization
3:     if M_U < T_LoJP then
4:         MARP ← current MARP value
5:         MARP ← MARP + inc
6:         RELOAD(MARP)
7:         SLEEP(delay)
```

ning in parallel. It therefore requires to better balance the resources allocated to M_{AM} and M_{YC}. Algorithm 6 applies another greedy heuristics to gradually reduce the amount of memory allocated to MRAppMaster by a floating step (*dec*) until one detects that the overall memory utilization (M_U) falls below the maximum memory utilization threshold T_{LoJT}.

Algorithm 6 Fixing LoJT by decrementing MARP.

```
1: procedure LoJT(T_LoJT, dec, delay)
2:     M_U ← current memory utilization
3:     if M_U > T_LoJT then
4:         MARP ← current MARP value
5:         MARP ← MARP − dec
6:         RELOAD(MARP)
7:         SLEEP(delay)
```

To avoid an oscillation between the two strategies, we combine them in a double-threshold (T_{LoJP}, T_{LoJT}, where $T_{LoJP} < T_{LoJT}$) heuristic algorithm that ensures that they work in synergy (cf. Algorithm 7). In the experiments, based on the experience, we set 0.9 and 0.95 as T_{LoJP} and T_{LoJT}.

The increment and decrement steps are not fixed. Instead, they are computed in each loop iteration based on the difference between the memory utilization and the target threshold. This allows the system to automatically achieve the translation between rapid and fine-gained tuning—*i.e.*, if the M_U is near a threshold, the square root will be small while shall the memory utilization be far from a threshold the increment or decrement will be large.

5.1.3.3 Handling Drops of Memory Utilization

Drops of memory utilization are caused by the completion map and reduce tasks that release large blocks of memory. Such memory fluctuation can result in

Algorithm 7 Balancing LoJP and LoJT.

```
1: procedure BALANCE(delay)
2:     M_compute ← overall maximum memory
3:     T_LoJP ← 0.9 × M_compute
4:     T_LoJT ← 0.95 × M_compute
5:     loop
6:         M_U ← current memory utilization
7:         if M_U < T_LoJP then
8:
9:             LOJP(T_LoJP, sqrt(T_LoJP − M_U) / M_compute, delay)
10:
11:        else if M_U > T_LoJT then
12:
13:            LOJT(T_LoJT, sqrt(M_U − T_LoJT) / M_compute, delay)
```

MARP oscillation when the algorithms above will be constantly scaling up and down the MARP value. To prevent this, we use a Kalman filter to smooth the input—*i.e.*, the memory utilization. It helps to stabilize the value and eliminate the noise induced by the memory fluctuation [45]. Concretely, we apply a 1D filter defined as:

$$M(t + \delta_t) \quad = \quad A \cdot M(t) + N(t) \tag{5.5}$$

where M refers to the state variable—*i.e.*, the memory usage—A is a transition matrix and N the noise introduced by the monitoring process.

5.1.4 Evaluation in Hadoop Cluster

In this section, we evaluate the capability of the self-balancing approach to address the problem MapReduce job parallelism and throughput. We start with an quick overview of the implementation of the self-balancing algorithm followed by a series of experiments. The evaluation has been done using a cluster of 11 physical hosts deployed on the GRID5000 infrastructure, the same as we used in Section 5.1.1. The Hadoop version is 2.6.1. Additional configuration details and experiment raw values are available in Appendix A.

5.1.4.1 Implementation Details

Figure 5.9 depicts the architecture of the feedback control loop that implements the balancing algorithm introduced in the previous section. It follows the classical

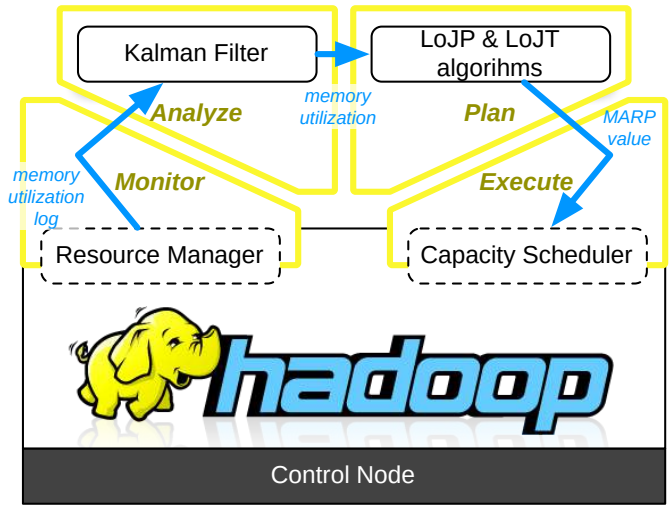


Figure 5.9: Architecture of the feedback control loop.

MAPE (*Monitor-Analyze-Plan-Execute*) decomposition [32].

This approach has 4 steps: Monitor, Analyze, Plan, and Execute.

(1) The Monitor step is responsible for reflecting the actual status of memory consumption across the whole Hadoop cluster. The memory information are collected using the ResourceManager services. Using ResourceManager services is much better than per-node monitoring method. Each NodeManager regularly reports on the state of corresponding compute node to ResourceManager by *heartbeat* (*i.e.*, the system message in Hadoop), such that ResourceManager can detect the total available resource in Hadoop cluster. Furthermore, each new *container* of applications should be directly launched (MRAppMaster) by, or negotiate (YarnChild) with ResourceManager. In such case, ResourceManager can record both of the latest memory usage of applications and the total available resource in Hadoop cluster, thus logs the total memory utilization into the system log file. To obtain the memory information of Hadoop cluster, the only cost is a reasonable delay (*e.g.*, seconds). In contrary, per-node monitoring method requires per-node sensor which might increase the burden of each compute node. Moreover, this method also will occupy the network bandwidth that in turn affects the Hadoop performance.

(2) The Analyze step contains a Kalman filter. As exposed in Section 5.1.2, the **Large Drops** is a set of fluctuations of memory utilization, which are caused

by the application self-managing mechanism introduced by YARN. These fluctuations are changeable, random, and violent in some cases. They disturb the balancing algorithms by misjudging the actual state of Hadoop cluster, thereby reducing the benefit of this approach, even making self-balancing approach launch unnecessary adjustments and degrading Hadoop performance. In this case, a Kalman filter is introduced into this approach to smooth the fluctuations, thus guaranteeing the effect of self-balancing algorithms.

(3) The self-balancing algorithms is the core of this approach, which are implemented in **Plan** step. Based on the smoothed input from Kalman filter, the algorithms will figure out the optimal MARP value, and require **Execute** step to refresh the parameter in Hadoop cluster.

(4) The **Execute** step is to update the MARP value and to reload it at runtime. The MARP value is accessed via YARN configuration and changes to it are applied using YARN resource manager admin client[4].

The control loop was implemented in Java and runs on the control node alongside with YARN. For the Kalman filter, we used the jkalman library[5]. We set the *delay* to 10 seconds before continuing next control loop iteration. We find that this is a reasonable delay allowing the system to apply the new configuration.

5.1.4.2 Job Completion Time

We start the evaluation by running the same set of MapReduce benchmark as we did at the beginning in Section 5.1.1—*i.e.*, WORDCOUNT, TERASORT and SORT from the HIBENCH benchmark suite, each with two datasets (30MB and 3GB). Figure 5.10 shows the mean job completion time of 100 jobs using (1) the vanilla Hadoop 2.6.0 configuration (MARP = 10%), (2) the *best-effort* statically profiled configuration where the values were obtained from the initial experiments (cf. Fig. 5.2), and (3) finally the self-balancing approach (dyn). The values were normalized to the vanilla configuration.

For each of the considered applications and workloads, the self-balancing approach outperforms both of the other configurations. Often, the difference between

[4] `yarn rmadmin` command

[5] `http://sourceforge.net/projects/jkalman`

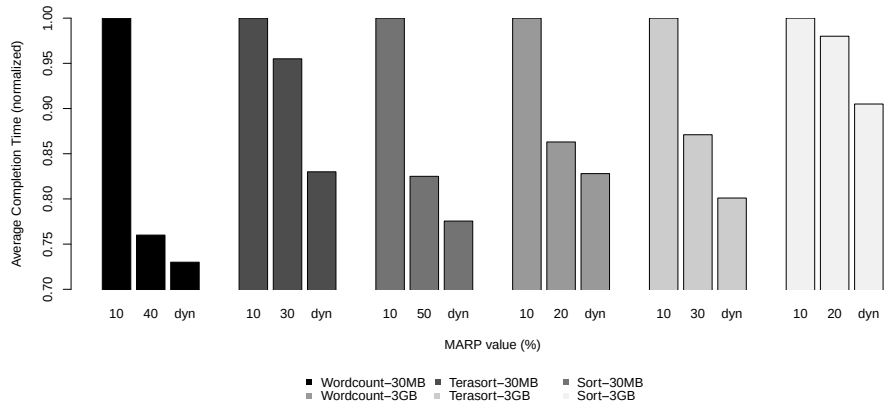


Figure 5.10: Performance comparisons of 3 HiBench applications and 2 datasets.

the statically profiled configuration and the dynamic one is small. This is because the *best-effort* MARP value already converges towards an optimal configuration so the applications cannot execute much faster. The important thing to realize is that the approach adapts to any application and does not require any profiling effort. It continuously finds a MARP configuration under which the application executes at least as fast as under the *best-effort* configuration.

Next, we evaluate how the approach performs under different workload sizes. Figure 5.11 shows the completion time of the Terasort with 3GB input data size benchmark under varying number of concurrently running jobs—*i.e.*, 10, 50, 100 and 150. In this case, the self-balancing algorithm outperforms the other configurations in all but the first case of a small number of jobs. The reason is that the solution always starts with the default MARP configuration which is 10% and converges towards the optimal value (20% in this case) along the execution. However, the overall completion time of the 10 jobs is too short and so the jobs finish before the algorithm converges.

Finally, we evaluate the approach with 4 time-varying workloads generated by SWIM. We use the same workloads as we presented in Section 5.1.1. The job size distribution varies across the different workloads. Each job has only one reduce task and a varying number of map tasks chosen randomly from a given map size set. The overview of the workload configurations is given in Table 5.1. Each map task manipulates (reads or writes) one HDFS block; in this case 64MB. The complete input size of the workload is shown in the last column.

Figure 5.12 compares the job absolute completion time for static and dynamic

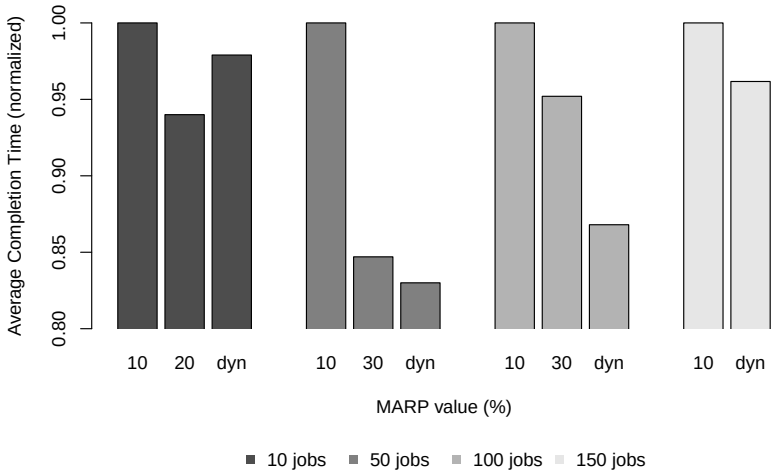


Figure 5.11: Performance comparisons of Terasort configured with 3GB under 4 workloads. The *best-effort* MARP for the case of 150 jobs is the same as the vanilla MARP—*i.e.*, 10%.

	#Jobs	#Maps	Map size set	Input size
W1	500	10460	$\{5, 10, 40, 400\}$	335GB
W2	500	25605	$\{5, 10, 50, 100, 300, 400\}$	819GB
W3	1000	5331	$\{1, 2, \ldots, 35\}$	342GB
W4	500	15651	$\{26, 27, \ldots, 50\}$	500GB

Table 5.1: Configuration of SWIM workloads.

configuration parameters. For each workloads, one can observe that, compared to the vanilla configuration, the approach can significantly reduce the completion time of jobs (*e.g.*, up to 40% in W1). It also systematically delivers a better performance than the *best-effort* configurations. Figure 5.13 reveals the detail of job completion time captured from W1. Compared to LoJP (Figure 5.13a) and LoJT (Figure 5.13b and 5.13c) cases, the dynamic configuration can maintain balance between the job parallelism and job throughput, resulting in the shortest job completion time for most of jobs (*i.e.*, the optimal performance). The approach can substantially reduce the waiting time, meanwhile, without increasing processing time significantly. The job completion time therefore is shortened, resulting in performance optimization. The overall completion times of the four SWIM workloads is further shown in Figure 5.14. Similarly to what has been shown in the previous figure, the approach outperforms all the other configurations.

Finally, for illustration in Figure 5.15, we show the different MARP values

computed over the processing of the workload W1 by the self-balancing algorithm. We can observe a weak correlation between this plot and the W1 job distribution shown in Figure 5.4. The reason why the correlation is weak is that the size of the individual jobs varies as we have shown in Table 5.1. The MARP value follows the dynamics of the jobs executions, therefore, it is also shifted in time as any load peak (in the number of job submissions) will be propagated in the system with delay.

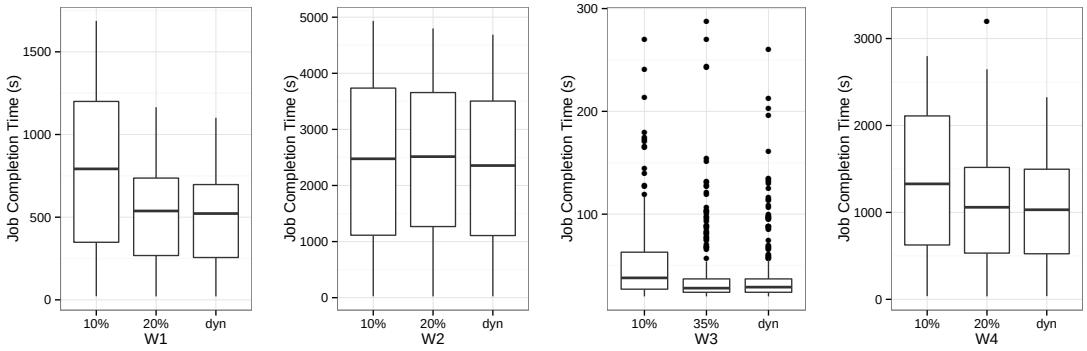
5.1.4.3 Job Resource Consumption

The approach focuses on balancing the memory allocation between MRAppMaster and YarnChild containers. In this section, we therefore evaluate if the memory is ideally used while running MapReduce jobs by computing the evolution of $Score_{t,w} = \frac{MU_{t,w}}{CT_w}$, where $Score_{t,w}$ is the ratio at time t for the workload w that we compute as the memory utilization ($MU_{t,w}$) divided by the overall completion time (CT_w). Figure 5.16 shows the value of $Score$ for the 4 SWIM generated workloads introduced in the previously introduced. We can observe that for each of the workloads, the approach gets a higher score than the best static configuration.

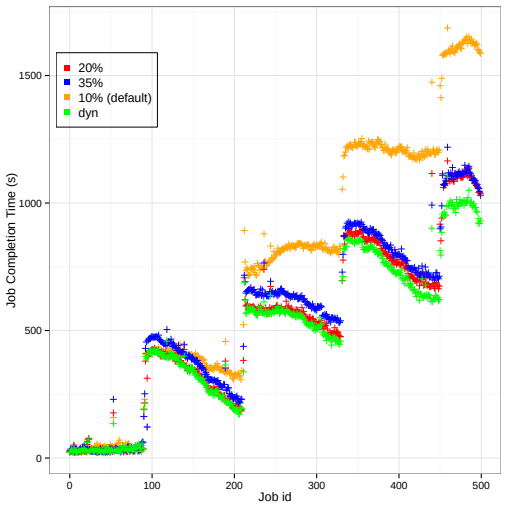
5.1.4.4 Discussion

Typically, the identification of an optimal static configuration can be either based on an experience or requires profiling techniques and several executions [38]. The approach instead automatically adjusts this value based on the current state of the Hadoop cluster.

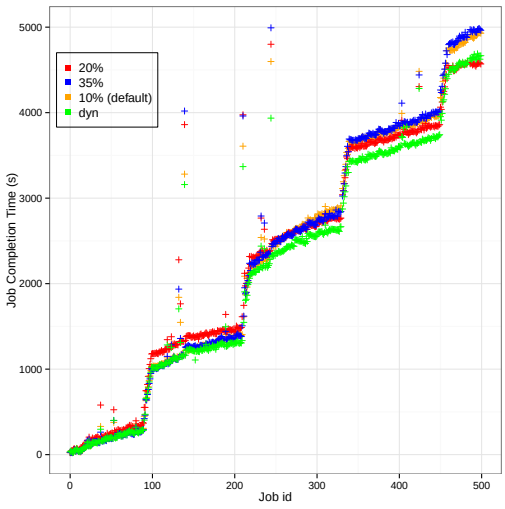
The evaluation shows that the self-balancing of job throughput and parallelism performs better than static configurations under various conditions of execution, without any prior application knowledge nor domain expertise. This solution works with standard Hadoop distributions as it only requires to access YARN for getting information about the current memory utilization and adjust the MARP configuration value.



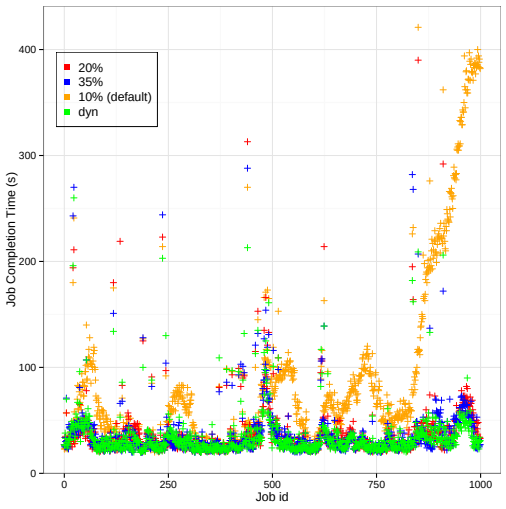
(a) The distribution of job completion time



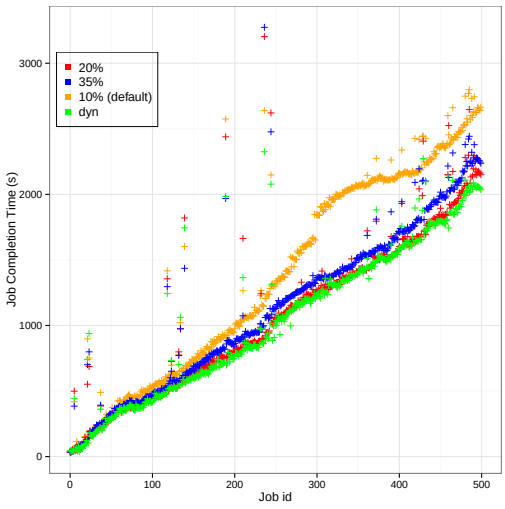
(b) W1



(c) W2

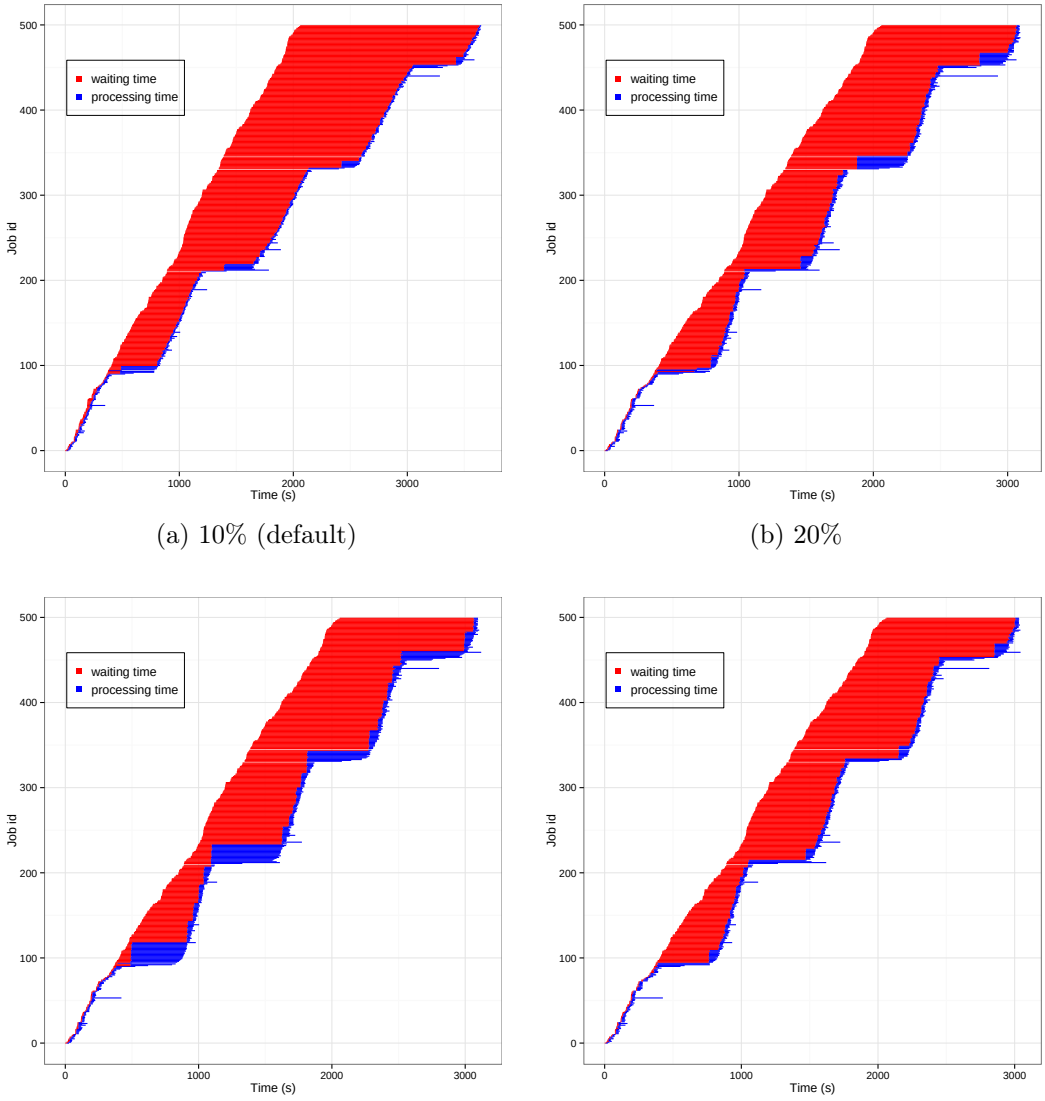


(d) W3



(e) W4

Figure 5.12: The comparison of job absolute completion time observed for static and dynamic configuration parameters.



(a) 10% (default)

(b) 20%

(c) 35%

(d) dyn

Figure 5.13: The detail of job absolute completion time of W1.

5.1.5 Summary of Self-balancing Optimization

Optimizing the performance of Hadoop clusters has become a key concern for Big Data processing. In YARN, inappropriate memory usage may lead to significant performance degradations. In this chapter, we propose a self-adaptation approach based on a closed feedback control loop that automatically balances the memory utilization between YARN MapReduce processes. We have shown that it outperforms the default Hadoop configuration as well as the *best-effort*, statically profiled, ones. Thanks to this self-balancing approach, Hadoop cluster can process majority of concurrent workloads with the optimal performance. Furthermore,

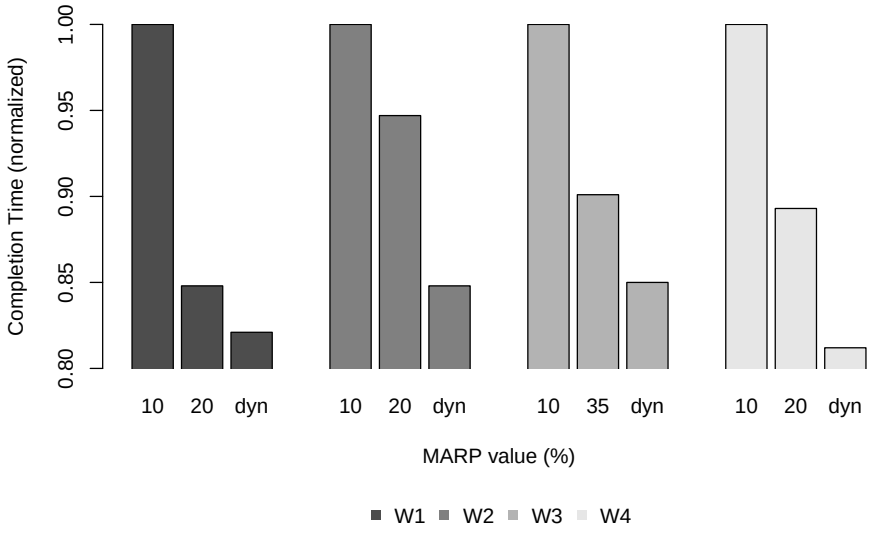


Figure 5.14: Performance comparisons of 4 SWIM workloads.

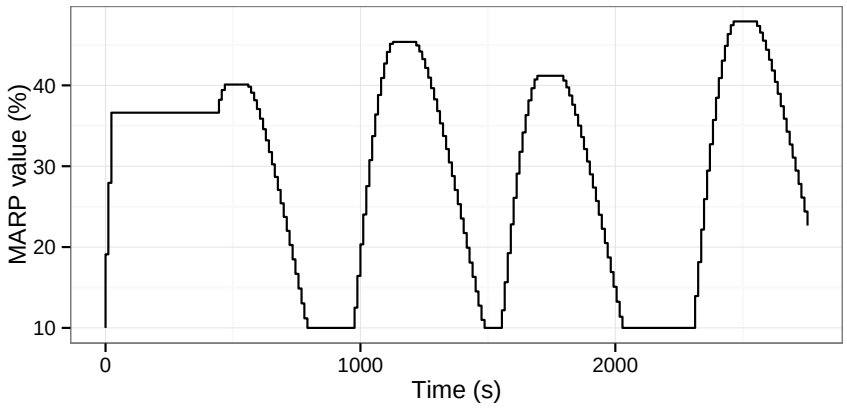


Figure 5.15: Computed MARP value by the self-balancing algorithm during the execution of W1.

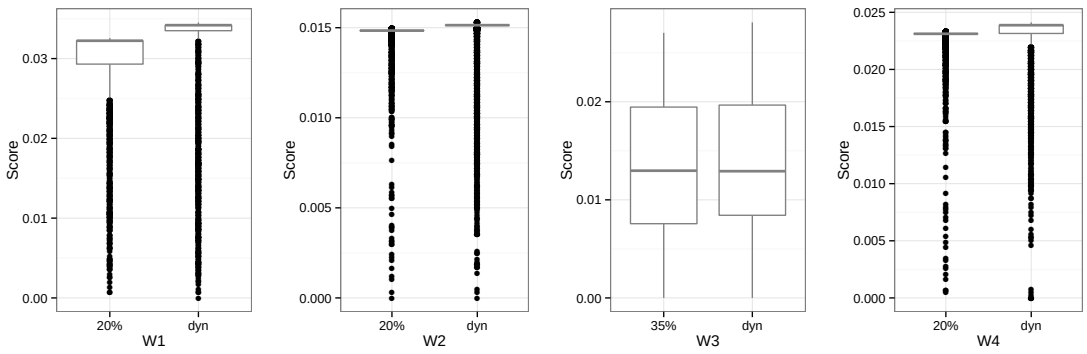


Figure 5.16: *Score* comparison of the 4 SWIM workloads.

while in this research we focus on MapReduce paradigm, the approach in fact works on YARN level. In this case, it can be used for all YARN-based applications and therefore we also plan to look for other applications based on YARN to evaluate this approach.

5.1.6 Synthesis

In platform level, this approach can ensure the optimal system performance for Hadoop by optimizing memory consumptions. However, when the Hadoop cluster has processed the congestion, most of the compute nodes will probably become idle. This is especially true when the input workloads are time-varying. The idle compute nodes would not cause any problem for an ad-hoc infrastructure. However, for a multi-tenant infrastructure, the occupation of idle resource (*i.e.*, compute nodes) will degrade the resource utilization, particularly in the case of Cloud computing.

Thanks to IaaS service, users can flexibly allocate resource from the Cloud. The Cloud provides its resource in the form of VM with high security and strict access—*i.e.*, the Cloud infrastructure is shared by multi-tenant but the provisioned VM is only available to its corresponding users. In the Hadoop case, the other Cloud-users cannot re-allocate the provisioned resource even though the Hadoop Cluster becomes idle—*i.e.*, the VMs of idle compute nodes in Hadoop cluster cannot be reused by another urgent requirements. This therefore degrades the resource utilization of the Cloud, resulting in high Cloud cost which might affects the lease price of IaaS.

Besides the performance improvement on platform level, we also focus on the resource management in infrastructure and propose a new middleware service (*i.e.*, CloudGC) to optimize the ossified mechanism of resource allocation in Cloud. Based on the experiments and analysis in an OpenStack Cloud, we can find that CloudGC is an interesting and feasible choice to avoid the resource leaks caused by resource allocation mechanism. The infrastructure we use is a private OpenStack Cloud which consists of 8 compute nodes. The optimization of resource management (*i.e.*, CloudGC) in this Cloud is explained and be proved in Chapter 4.

We believe that, thanks to the resource self-optimization approaches in both platform- and infrastructure-level, Cloud providers can easily guarantee the performance of its high-level platform (*i.e.*, Hadoop cluster), while *(i)* degrading the impact of idle compute nodes in the Hadoop cluster and *(ii)* improving the resource utilization of Cloud—*i.e.*, releasing the resource when the Hadoop cluster becomes idle—to avoid unnecessary growth of Cloud expenditure. Therefore, we can also consider that these solutions are important complement to the elasticity researches in their respective level.

After the researches on resource optimization of the provisioned infrastructure, we also focus on the deployment and resource provisioning at the platform level (*i.e.*, Hadoop cluster). For the elasticity researches on Hadoop, the timeliness of provisioning resources as well as rapid deployment of platform is an important issue. Moreover, the reproducibility is also a problem bothering many Hadoop researchers for a long time when evaluating their works (*e.g.*,, self-adaptation approaches), due to the diverse Hadoop performance on different infrastructure. Therefore, with regards to the various problems (*i.e.*, timeliness and reproducibility) about (rapid) deployment of Hadoop cluster, we develop a new tool to help users to quickly deploy a complete Hadoop cluster on their infrastructure, with the same default configuration each time. In next section, we will introduce this rapid deployment tool of Hadoop cluster, which is based on docker.

5.2 Rapid Deployment of Hadoop Cluster

In the typical researches of distributed systems, elasticity plays an important role. It is certainly an important and effective solution for Hadoop cluster to optimize its performance. In this research, we focus on the rapid deployment of a complete Hadoop cluster, which is an indispensable part of Hadoop elasticity. One of the characteristics of many Hadoop workloads is that their dynamics changes over time. Considering the scale of these workloads (thousands to hundreds of thousand jobs), this can quickly lead to a waste of resources since the static configuration does not adapt to the current runtime condition.

Optimizing Hadoop execution has therefore attracted a lot of research attention, resulting in a number of different approaches in particular in the domain

of self-adaptive software systems [8, 14, 22, 28, 36, 38, 47, 48]. However, the research effort is often hindered by the accidental complexity of (1) setting Hadoop deployment in different distributed environments and (2) comparing the effect of different approaches.

Hadoop is a highly distributed systems which contains hundreds of configuration parameters. Correctly setting an operational Hadoop cluster requires a significant amount of system administration knowledge and effort. Currently, there is no easy way to share and reproduce experimental evaluation of Hadoop-related researches, such as the existing self-adaptive approaches of Hadoop (*e.g.*, the self-balancing approach introduced in Chapter 5.1). Furthermore, the Hadoop performance also highly depends on the infrastructure where the cluster deploys. It is therefore rather complex to compare the effect of approaches from one to another since the experiments are hard to reproduce (*e.g.*, re-creating a testbed similar to the one used in their experiments, availability of the implementations). This makes the Hadoop researchers to have to find out an tool which can help them to easily and quickly reproduce an operational Hadoop cluster on their infrastructure.

In this chapter, we address these limitations by proposing *hadoop-benchmark*, an open-source research acceleration tool for rapid prototyping and evaluation of self-adaptive behaviors in Hadoop clusters. The main objectives are
(1) *rapid prototyping*, allowing researchers to quickly begin to experiment with self-adaptation approaches in Hadoop, without the needs to cope with low-level system as well as infrastructure details, and (2) *reproducibility*, allowing researchers to share complete experiments for others to experiment with and to compare them with their work. This tool achieves this by providing the following features: (1) a declarative mechanism to provision complete (configured and running) Hadoop cluster on either a local machine, local cluster or in a number of cloud providers (*e.g.*, Google Cloud Engine, Microsoft Azure, Amazon AWS), and (2) a number of pre-configured, well-known Hadoop benchmarks to easily assess the cluster performance. The cluster deployment and benchmark execution are done in an automated way based on simple configuration files, which can be easily shared. The provisioned nodes in the Hadoop cluster further includes monitoring service that can be used for developing *touchpoints* for system identification and the mon-

itoring part of feedback control loops which governs the self-adaptation.

To demonstrate the usage of *hadoop-benchmark*, we include a complete implementation of a Hadoop self-adaptation case study. Concretely, a feedback control loop that balances Hadoop job parallelism and throughput through Hadoop capacity scheduler adjustment—*i.e.*, one implementation of the self-balancing approach (based on classical feedback control loop), which is introduced in Chapter 5.1 is also packaged in this tool.

It is important to note that, while Hadoop has been mostly connected with implementation of MapReduce paradigm, it has grown and, since version 2, it has become a general framework for distributed large-scale applications. The focus on Hadoop goes therefore beyond MapReduce and has wide applications to other technologies that are based the core enabling technologies—*i.e.*, distributed files-systems (*e.g.*, HDFS) and application scheduler (*e.g.*, YARN).
This artifact is therefore available from:

https://github.com/Spirals-Team/hadoop-benchmark

5.2.1 Motivation

Hadoop has become a famous data-processing distributed systems for couple of years. Many researchers have proposed various approaches to optimize its performance as well as many well-known benchmark suites. This provides lots of choices for later researchers to evaluate their work. However, due to the complexity of Hadoop and the mutable performance which highly depends on its infrastructure, researchers have to face a new problem. That is, while experimenting with self-adaptation in Hadoop cluster, a researcher must perform a number of recurring tasks which include setting up a testbed, running experiments, extracting data from computing nodes. All of these tasks are both time consuming and requiring significant amount of domain-specific knowledge, to operate all the software stacks involved.

Due to the repetitive nature, researchers often develop numerous *ad hoc* scripts to automatize various tasks, but these are usually prone to error. Therefore, keeping the experiment infrastructure in a desired state requires a lot of manual effort. Furthermore, it complicates reproducibility of the experiments and limits

the possibility to share and compare results.

In this case, many researchers therefore need a tool that will automatize these tasks and share the experiments so that they can be easily reproduced by others. Concretely, it should support:

1. *Automated and rapid deployment of complete Hadoop cluster.* Based on a single configuration that can be stored in version control, this tool should be possible to deploy a complete Hadoop environment in a number of cloud providers as well as on a local machine. The deployment should be fast to reduce the time required for evaluating new approaches.

2. *Automated benchmark execution.* It should be possible to execute the common acknowledged Hadoop benchmarks (*i.e.*, `hadoop-mapreduce-examples`[6], HiBench[7], Swim[8]) to evaluate the performance of a given Hadoop-based approach in different research area, such as self-adaptive domain.

Beyond the above features, the tool should be designed with the flexibility—*i.e.*, the tool should have a good adaptability in different environments and the interface for future development. It should allow one to experiment with the various aspects of Hadoop systems, to provide hooks for the different life-cycle phases on the Hadoop components and the possibility to develop required sensors and effectors.

5.2.2 Docker Container Technology

This section provides a brief overview of the core enabling technology of *hadoop-benchmark*—*i.e.*, Docker.

Docker is an open-source project that aims at automating application deployment. It achieves that through a concept of software containers, a layer of abstraction built on the top of system-level virtualization offered by Linux operating system. Essentially, a container offers a process-level resource isolation where each container has its own address space, file system and networking. Containers do

[6] `https://github.com/apache/hadoop/tree/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples`

[7] `https://github.com/intel-hadoop/HiBench`

[8] `https://github.com/SWIMProjectUCB/SWIM`

not need any hypervisor (like in the case of classical virtualization), they are simply operating-system-level processes, yet completely separated from one another. A host machine can therefore run a number of them concurrently.

A container is based on an image, a template that contains all the resources a container needs. They are created from base images using a simple domain-specific language describing a sequence of simple instructions (*e.g.*, execute a command, add a file) that allows one to tailor the image to one's needs. For example, in this case, these steps include downloading Hadoop distribution and its basic configuration. From an image, a container is created by a docker daemon (cf. Fig. 5.17). It instantiates the image, allocates a file-system and network interface, sets up an IP address and performs other tasks to bootstrap the container.

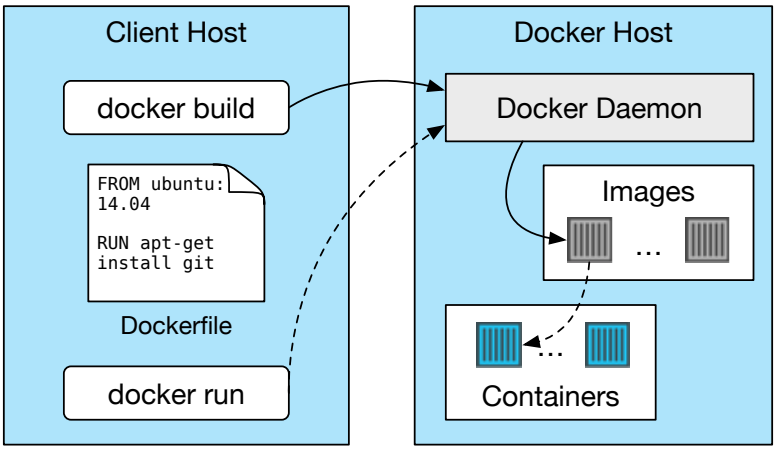


Figure 5.17: High-level Docker architecture.

The main advantage of docker versus classical virtualization is in the reproducibility and ability to deploy across a range of systems. A container definition is composed of a set of configuration files that can all be stored in a version system. It is also possible to create a binary build of an image share it via the DockerHub[9] infrastructure to save on building time. A docker image is also technologically agnostic and can be run on any Docker host. This is not the case with virtual machines technologies (*e.g.*, it is not possible to run Virtualbox image in Amazon EC2). Finally, spawning a container is an order of magnitude faster than spawning a virtual machine (since there is no need to boot a new operating system) [21].

[9] `https://hub.docker.com`

This contributes to automation instead of doing changes manually in an *ad hoc* manner.

5.2.3 Overview of *hadoop-benchmark*

The Hadoop research acceleration tool, named *hadoop-benchmark*, is based on the Docker container technology. Essentially, it is a set of Docker images and a script that drives the deployment and the execution of Docker hosts and containers, which are deployed on these hosts. To improve the adaptability, this tool is designed to only contain a small code base with minimal dependencies. Besides the enabling technology—*i.e.*, Docker and Docker Machine—the tool therefore only requires bash and git.

There are three types of images: (1) *base images* that provide a vanilla Hadoop installation, (2) *extensions to the base images* with custom configuration coupled with implementation of some self-adaptive behavior, and (3) *benchmark images* that each executes a particular benchmark suite. Currently, we provide one base image (which is split in two to shorten the download time, furthermore users can also customize the base image from the intermediate image to save build time), one extension image implementing the case study (cf. Section 5.2.4) and three benchmarks: `hadoop-mapreduce-examples`, HiBench and Swim. Since the project is publicly available on GitHub, any contribution in forms of pull requests is welcomed. We hope that this will help to gradually extend the number of scenarios and benchmarks provided in order to foster the evaluation of self-adaptive Hadoop solutions. All the images are presented in two forms, a source form in the GitHub repository and a binary form in the DockerHub repository. The latter can be used to bypass manual image build and reuse the binary version that can be automatically downloaded by Docker daemon.

The base image consists of a minimal Ubuntu operating system with Java and a vanilla Hadoop distribution. The only Hadoop settings we provide cover networking making sure all the Hadoop components can communicate with each other.

Before any of Docker containers can be launched with these images, it is necessary to form a cluster of Docker hosts to host these Docker containers. The tool

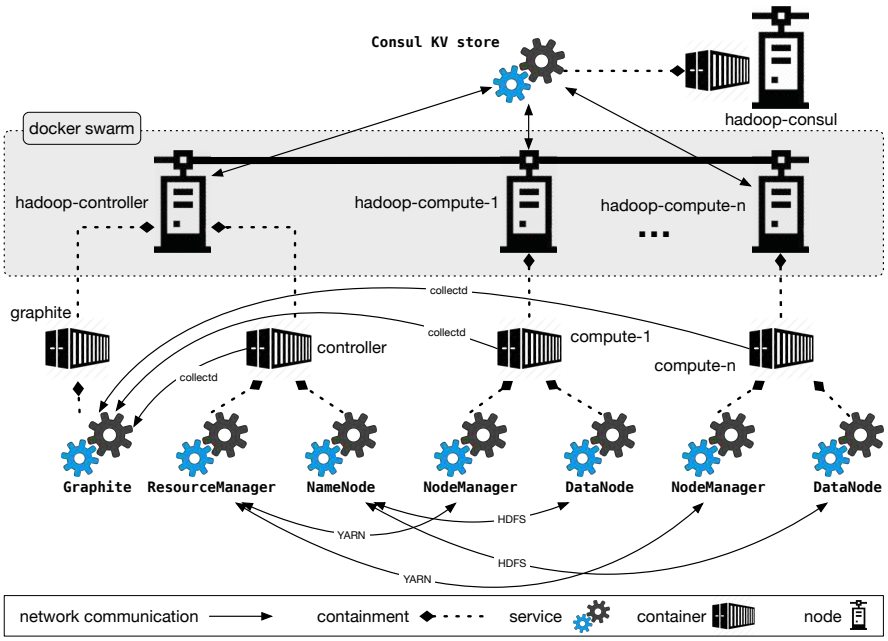


Figure 5.18: High-level overview of the hadoop-benchmark-platform provisioned cluster

facilitates this process with a set of commands that manage cluster life-cycle—*i.e.*, start, stop, restart, destroy. Based on a simple configuration (a number of nodes and details about the environment where the nodes shall be created), it can provision virtual machines with Docker Machine support. All the virtual machines will be also connected in a virtual network (*i.e.*, Docker Swarm Network) so the containers that are hosted in different Docker hosts can still communicate with each other. All these works rely on *Docker Machine*[10], an official Docker service that is used to create Docker hosts. It currently support local deployment using VirtualBox or VMWare as well as a number[11] of cloud providers including all the major vendors. It also allows one to connect existing hosts in local clusters through a generic SSH connection. The main advantage of using Docker Machine is that, in the layer of abstraction, it can always form the same infrastructure for Hadoop cluster, regardless the actual virtualization or clustering environment. From future perspectives, the *hadoop-benchmark* further supports deploying multiple clusters where multiple experiments can therefore run in parallel.

Figure 5.18 delivers a high-level architecture overview of a provisioned Hadoop cluster by *hadoop-benchmark*. Each cluster contains the following nodes:

[10]https://docs.docker.com/machine
[11]A list of supported environments: `https://docs.docker.com/machine/drivers`

– One `hadoop-consul` runs a single Docker container with the Consul service[12], a distributed key-value store used for service discovery. Concretely, it is used as a background mechanism of the virtual network (*i.e.*, Docker Swarm network), which is formed when the Docker hosts are created. This node is the core of Docker Swarm network which guarantees the communication among the Docker containers hosted in different Docker hosts.

– One `hadoop-controller` acts as the controller node for the Hadoop cluster. It runs two containers: `controller` and `graphite`. The former provides the ResourceManager and NameNode services. The latter contains the Graphite service[13] for real-time visualization of monitoring data (*e.g.*, CPU, memory, I/O) coming from the other containers.

– A variable number of `hadoop-compute` nodes represent the Hadoop compute nodes. They run a single container with NodeManager and DataNode services.

The `hadoop-contoller` and `hadoop-compute-*` nodes are connected into *Docker Swarm*, a native clustering mechanism for Docker. *Docker Swarm* allows all the connected nodes to become a part of the same virtual network and to communicate with each other. Each node in the cluster is further equipped with a monitoring service, *collectd*[14]. It collects a common set of performance-related metrics and make them available in CSV and RRD[15] format. They can be easily fed into any monitoring part of a feedback control loop.

A such deployed Hadoop cluster has no difference with a complete Hadoop cluster, which is directly installed on physical infrastructure. All the acknowledged Hadoop benchmarks can be used to assess the performance of a Hadoop cluster deployed by *hadoop-benchmark*. In the tool, each Hadoop benchmark is implemented as a Docker image and runs in a standalone container alongside the `controller` container. The tool also allows to quickly access logs of any of the services from the client machine as well as to access to a shell inside any of the running containers. Moreover, all the data from the containers are available to be

[12] `https://www.consul.io`
[13] `http://graphite.wikidot.com`
[14] `https://collectd.org`
[15] `https://en.wikipedia.org/wiki/RRDtool`

mounted on a client machine or to any additional containers created within the same network.

5.2.4 Case Study

As a use case to show the capabilities of *hadoop-benchmark*, we choose a self-adaptive behavior that automatically balances the job parallelism and throughput in a Hadoop cluster introduced in Chapter 5.1. Concretely, it adjusts a YARN capacity scheduler parameter, MARP[16], which controls the ratio between the number of concurrently executing MapReduce jobs versus the number of running map and reduce tasks. This is one of the crucial parameter whose inappropriate configuration can have a serious impact on Hadoop performance (up to 40%).

The proposed tool accelerates the implementation in the following ways. First, it helps us to validate the hypothesis—*i.e.*, the impact of MARP on the cluster performance. The tool quickly deploys a Hadoop cluster and allows us to run a series of experiments using the standard Hadoop benchmarks (PI ESTIMATION). For each experiment, we change the MARP value and observer its effect. The results are shown in Figure 5.1b.

Once the hypothesis is validated, we can start prototyping the actual implementation. To make this easily reproducible, we create a new docker image that extends from the base image and include the implementation code together with a start-up hook. Concretely, we create a feedback control loop in Java that periodically observes the cluster memory usage and based on its value, it proportionally adjusts the MARP value. It uses a proportional controller coupled with a Kalman filter to control the MARP value. The start-up hook is a shell script that launches the Java application after the ResourceManager is started. The cluster memory usage is extracted from the ResourceManager log file to which all NodeManager reports periodically their memory statistics.

To tune the controller, the same series of benchmarks and rerun while experiment with different controller settings. We can leverage from the ability to execute experiments in multiple clusters in parallel to saves time in cases of long running experiments.

[16]Maximum Application Master Resource in Percent

The performance gain can be compared by simply running the experiment twice each time with different docker image—*i.e.*, either the base image with vanilla Hadoop installation or Hadoop with our self-balancing controller. Shall there be any other approach we would like to compare with, we could have run it in the very same manner. A sample result from Hadoop cluster made of 11 physical hosts[17] (1 control node and 10 compute nodes) deployed on the GRID5000 infrastructure [18] is shown in Figure 5.19.

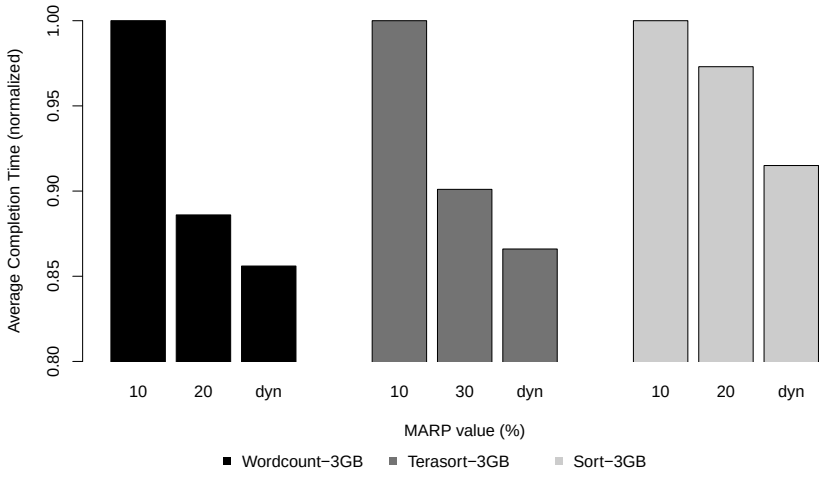


Figure 5.19: Performance comparisons of 3 HIBENCH. The first bar corresponds to the vanilla configuration—*i.e.*, 10%—the second to the best statistically profiled value, and the last to our self-balancing approach.

Finally, since the cluster configuration is saved in the version control together with the description of the docker image, anyone can reproduce these experiments in their infrastructure. To do that, one only needs to clone the repository, possibly adjust the cluster configuration to fits one deployment and rerun the benchmarks.

5.2.5 Assessment

This work has been driven by the need of a tool that can be used to rapidly deploy Hadoop cluster in various infrastructure, to prototype diverse Hadoop-related approaches (*e.g.*, self-adaptation approach) in Hadoop clusters and allows

[17] 2 Intel Xeon L5420 CPUs, 4 cores, 15GB RAM, 298GB HDD
[18] `http://grid5000.fr`

one to easily share or reproduce the experiments and results. The two main objectives were to provide (1) an automated way to rapidly *deploy Hadoop clusters*, and (2) an automated way to *execute Hadoop benchmarks*. Both with a focus on reproducibility, *hadoop-benchmark* not only helps users to quickly tune Hadoop cluster but also reproduces a same environment. We do believe this is interesting and important for the Hadoop researchers who want to share their approaches or compare them with others.

The automated deployment of Hadoop cluster is facilitated by the orchestration of Docker Machine and Docker containers. This allows us to deploy Hadoop transparently in a number of environments ranging from local machine, local existing cluster to major cloud providers. All deployments are based on the same simple configuration which is stored in a version control. Despite that there is some overhead induced by creating virtual machines in the case of local deployment or deployment in cloud, this only needs to happen once in the first time when docker-machine must install docker services on the infrastructure. The actual deployment of Hadoop cluster is based on docker containers with prepared docker images. Furthermore, spawning a container is close to spawning a regular operating system process—*i.e.*, the overhead is minimal. This allows us to deploy a complete new Hadoop cluster in just a few seconds. This is considerably faster than the other solutions that require full redeployment of virtual machine (in order to make sure there are no stalled data from previous deployment) which may take from dozens of minutes to hours.

The automated benchmark execution is similarly based on docker containers. Currently, we provide three well-known Hadoop benchmarks:

`hadoop-mapreduce-examples`, HiBench and Swim.

All steps from assembling the cluster to executing a benchmark is driven by configuration files and few bash scripts with no other dependencies. Any changes can therefore be kept in plain text files that are be stored in version control. The project is open-source and the complete source code is hosted on GitHub: `https://github.com/Spirals-Team/hadoop-benchmark`.

5.2.6 Summary of *hadoop-benchmark*

This chapter reported on the implementation of an open and reproducible tool, named *hadoop-benchmark*, which can rapidly deploy a complete Hadoop cluster for the prototyping, several well-known benchmarks and the evaluation of self-adaptive behaviors in Hadoop clusters. While Hadoop is acknowledged as a *de facto* standard for the processing of large-scale dataset, the performances of Hadoop clusters tend to be affected by the underlying infrastructure as well as the considered workloads and algorithms. Optimizing Hadoop performance has therefore attracted a lot of research attention, in particular in the domain of self-adaptive software systems [8, 14, 22, 36, 47, 48]. Yet, reproducing and assessing the proposed contributions might quickly be hindered by the accidental complexity of Hadoop deployments.

This tool therefore leverages the state-of-practice in lightweight virtualization techniques to deliver a flexible approach to facilitate the researches in the software engineering of self-adaptive Hadoop systems. In particular, this chapter demonstrate this approach on the design and the implementation of a feedback control loop that autonomously adjusts the YARN capacity scheduler to appropriately balance the MapReduce jobs throughput and parallelism. Beyond this demonstration, we believe that this research asset can benefit the research community by providing a common environment to empirically compare the research contributions of Hadoop, particularly in the area of self-adaptive MapReduce applications. The design and implementation choices we made leverage the reuse and the extension of this environment in order to fit a large diversity of scenarios.

Part : Conclusion

Chapter 6

Conclusion

In this thesis, we focus on the platform- and infrastructure-level resource management in Cloud computing. Based on the diverse resource leaks caused by misconfiguration and stiff system mechanisms, two different solutions are proposed to improve the resource utilization in both layers—*i.e.*, platform and infrastructure.

Most of current researches adopt elasticity as the primary manner to dynamically tune system resources to guarantee the system performance keeping in a reasonable level. However, beyond scaling the infrastructure, our propositions in this thesis prefer to figure out the optimal performance based on provisioned resources, or maximize the resource utilization to serve more users as far as possible. Therefore, we believe that the results of researches in this thesis can be regarded as complements to the elasticity researches, both in infrastructure- and platform-level. The contributions of all researches described in this thesis will be further summarized in below sections, respectively.

6.1 Cloud Computing

Cloud computing gives its users an impression that it can be regarded as an unlimited pool of virtual compute resources, that shares the resources in multi-tenant. As a resource provider, it provides different models "as a Service" for users. The three basic models are Infrastructure-, Platform-, and Software-as-a-Service corresponding to hardware, environment, and application layers. IaaS does not only reduce the upfront expenses for hardware, but it also leverages the users from the maintenance of infrastructure. It allows the users to easily manage their infrastructure—*i.e.*, they can flexibly allocate resources to dynamically accommo-

date their requirements thus avoiding *over-provisioning* and *under-provisioning* problems—and also liberates them from complicated maintenance of infrastructure thereby enables the users to focus on their work. PaaS constructs a complete processing environment for its users. In this environment, the users do not need to take care of any issue on the platform, but just handle and develop their applications using the offered APIs. Similar to PaaS, SaaS supplies the running applications to the users, which have been well configured. Users do not have to take care of the operation and maintenance of the applications and, therefore, supporting such applications does not require any specific experience. Thanks to these benefits, Cloud computing has become a popular concept both in academia and in industry communities. In this case, more and more developers and entrepreneurs transfer their business to Cloud computing. This leads to prosperity of Cloud computing business, but also presents many new challenges to Cloud computing technologies, particularly in the case of private Cloud.

As a popular open-source framework for deploying Cloud computing on local machines, most of private Cloud are supported by OpenStack. OpenStack is a complex project consisting of diverse components where each of them focuses on one specific services, such as *Keystone* (Authentication service), *Glance* (Image Management), and *Neutron* (Network connections) etc. Among these components, *Nova* is the most important one which concerns the resource allocation and management in OpenStack Cloud—*i.e.*, All the VM instances in the Cloud are provisioned and managed by this component. The experiments in Section 4.1 report on a problem about resource utilization of Cloud infrastructure. That is, OpenStack (*i.e.*, *Nova*) ignores the actual resource utilization in Cloud when allocating resources to VM instances required by users. In other words, when the total user requirements reach the Cloud infrastructure limitation, the users cannot deploy any more VM instances no matter how the actual resource consumption is in the Cloud infrastructure. To optimize the resource consumption and provide the available resources for the other urgent requirements, the only solution is to delete the *idle* VM instances from OpenStack instance manager, manually. In this case, the Cloud administrators must continuously monitor each VM instance and recycle them when needed. This cannot obviously improve the QoS of Cloud

when its infrastructure become overcrowded, but significantly increases the workload of Cloud providers. Moreover, for the deleted VM instances, neither Cloud providers nor OpenStack have prepared an effective mechanism to recover them when they are required again.

To prevent these issues continuing bothering Cloud administrators and end-users, we propose a new middleware service, named CloudGC, in Chapter 4 to address these limitations of OpenStack.

- The first contribution of CloudGC is to solve how to accurately detect and determine the *idle* VM instances for recycling. OpenStack defines a set of states to distinguish various lifestyles of VM instances, which can be divided into two categories : *active* and *suspended*.

  The *suspended* VM instances remain in a dormant state, such as *paused* and *interrupted*. Users cannot do any actions in these VM instances unless wake them up. However, even though these VM instances cannot be used, they still occupy the resource quotas and prevent other requirements re-allocating the resources. The *suspended* VM instances can be considered as *explicit idle* VM instances. They can be directly indicated as *recyclable* by CloudGC and are snapshotted for recovery process, due to its inaccessible property.

  The *active* VM instances are always *running* in Cloud. That means, these VM instances are processing workloads or ready to process. In this case, the resource consumption of these VM instances is not stable and can be affected by end-users at any time. But among them, there are some VM instances which have waited for processing workloads in a long time. Therefore, these *running* VMs can be regarded as *implicit idle* VM instances with low CPU activity in a long enough idle duration. Even though these *implicit idle* VM instances have been inserted into the *queues*, users can still freely access to them and change their situation. In this case, the snapshots may need to be updated frequently, thus making network and system suffer from extra burdens, so CloudGC gives up snapshots for *implicit idle* VM instances and backs them up when needed.

  Furthermore, the two categories (*i.e.*, *explicit and implicit*) of *idle* VM instances are ordered by their duration of idleness. Therefore, we can ensure that the most

idle VM instance (*i.e.*, the one with longest idle duration) will be recycled firstly.

- The second contribution is to automatically decide on how to recycle the identified VM instances while guaranteeing they can be recovered. As reported before, the only way to free the occupied resources is to delete their corresponding VM instances. In this case, the recycling process has to face 2 problems: (1) how many VM instances should be recycled in heterogeneous Cloud, such as OpenStack Cloud, (2) how to recover these recycled VM instances when they are required again (this is truly possible because these VMs are not officially removed by users themselves).

  For the first problem, we developed an incremental approach within the recycling algorithm. When CloudGC receive a new request to provision VMs, it will first check the available resources in Cloud infrastructure. If the resources are not enough to support the new request, recycling process is invoked to recycle *idle* VM instances for collecting needed resources. Recycling process calculates the total resource requirements of new request firstly, based on the flavor (*i.e.*, the template of VM type defined in OpenStack) of new VMs and other informations. Then, following the *explicit first, most idle first* rule—*i.e.*, the recycling rule of CloudGC, the identified *idle* VM instances will be extracted from two *idle* queues one by one and be added into a temporary list. Meanwhile, during each extraction, the total amount of resource occupancy of the whole list will be accumulated, and be compared to the total resource requirements until Cloud has enough resources to support new request. Finally, the selected *idle* VM instances are removed from the *idle* queues, are snapshotted (if needed), and are recycled. If the two *idle* queues are exhausted while the recycling is insufficient, CloudGC will require OpenStack to reject the request or to apply other elasticity services to obtain additional resources.

  For the second problem, we propose a hook (named *ghost instance*) to bind floating IP of recycled VM instances on Cloud instance manager. When a VM instance is removed from Cloud instance manager, its occupied resources are released as available. However, in this case, the recycled VM also becomes unavailable. To guarantee that the recycled VMs are still alive for outside users

and free their resources at same moment, we extract their floating IPs and bind to the Cloud instance manager while recycling them. In this case, even though the *idle* VM instances have been removed from the Cloud infrastructure, the Cloud instance manager will replace the VM to continue receiving the requests, thus making the recycled VMs look like alive for external while they do not exist anymore in Cloud. Thanks to the *ghost instance*, CloudGC is able to recover the recycled VMs before forwarding the incoming requests to them.

- The last contribution allows CloudGC to recover the recycled VM instances. When the Cloud instance manager find some requests, which are sent to *ghost instances*, it should re-provision the recycled VMs to serve these new requests. The recovery process is similar to a new VM provisioning process, but the only difference is that, for the recovery process, the VM image is replaced with the snapshots of recycled VMs. Moreover, in an overcrowded Cloud, recovery process may also trigger another recycling process to collect available resources. This requires the CloudGC to support the iterative process.

  Even though the recovery process can ensure the reproducibility and availability of recycled VM instances, it is not an instantaneous process, but requires seconds to minutes for re-provisioning the VMs. Therefore, for some critical VMs, they cannot be recycled in any case. To support such exceptions, we propose a special list—*i.e.*, *pinned VMs list*—to mark these instances. Those VM instances that appear in this list will be ignored by recycling process and never be considered as recyclable targets. Furthermore, thanks to the coordination between recovery process and a time table service, CloudGC can realize "switch service" to make different groups of diverse-periodic VMs share the same Cloud infrastructure.

Based on the above challenges, we propose CloudGC—*i.e.*, a new middleware service aiming to eliminate resource leaks in Cloud computing, especially in the case of private Cloud. The current implementation is achieved by Python within OpenStack. The evaluations in Section 4.4 shows that CloudGC is able to help user requirements to exceed the Cloud infrastructure limitation. This guarantees that the Cloud resources will be allocated to real usage when needed, rather than be occupied for idleness.

6.2 Big Data

Big Data is a term for various operations and researches on large and complex data sets. Its topics consist of analysis, capture, search, sharing, storage, transfer, parallel processing, and information privacy etc. Due to the rapid growth of data sets gathered by numerous sensors within various environments, like large scale distributed systems or wireless networks, the data processing and management in Big Data require a set of techniques and technologies with new forms of integration to ensure and improve the performance and QoS. Moreover, these integrations can be described by 4V: (1) VOLUME presents the large amount of data sets, (2) VELOCITY reveals the fast speed of data processing, (3) VARIETY shows the wide range of data types, and (4) VERACITY exposes both the quality of data and the accuracy for finding target data. In this case, to accommodate these requirements and to facilitate the business and development of Big Data domain, a distributed data-processing platform (*i.e.*, Hadoop) is proposed under Apache License 2.0. At the moment, Hadoop has become the *de facto* data processing platform in Big Data domain. Since 2012, Hadoop has undergone an overhaul and introduces a new component YARN.

6.2.1 Self-balancing Algorithm

Hadoop is a well-known distributed data-processing environment and can be regarded as the *de facto* standard for Big Data processing. It is a general designation of a complete ecosystem in the Big Data domain. Hadoop supports various high-level paradigms and applications for diverse data-processing services, such as Hadoop MapReduce for batch processing, Pig for script, Hive for SQL services, etc. Besides the high-level services, Hadoop also contains two components to achieve cluster management and distributed data storage across sets of compute nodes. YARN is the component in charge of cluster management, which can be separated into two parts: cluster monitoring and job scheduling. For cluster monitoring, YARN proposes a master-slave distributed architecture consisting of two system-level subcomponents to gather machine informations from each compute node—*i.e.*, this architecture contains one ResourceManager to gather and record cluster informations from set of per-node NodeManager by Hadoop heartbeats (*i.e.*,

Hadoop system messages). The job scheduling relies on another subcomponent called container. Container has two types: AppMaster and YarnChild. AppMaster is the first container of its corresponding job, which is launched by ResourceManager. After the AppMaster is launched, ResourceManager transfers the job to this container for processing preparation. Then, many YarnChilds will be launched and managed by AppMaster to process the tasks of their corresponding job. In this case, the AppMaster and YarnChilds constitute a new job-level master-slave architecture. HDFS is responsible for the distributed data storage in a Hadoop cluster. It contains two subcomponents: NameNode and per-node DataNode. NameNode only stores the metadata of all data sets in the cluster. This can accelerate the data access by avoiding browsing the whole Hadoop cluster to locate the target data. And all the actual data is placed in compute nodes and is managed by corresponding DataNode. Even though YARN is able to achieve its objectives, it still may suffer from diverse problems, thereby degrading the Hadoop performance. In YARN, there is a subcomponent called CapacityScheduler containing a parameter MARP. This parameter manages the amount of memory in Hadoop cluster, which can be used for running AppMasters, thus controlling the number of jobs running in parallel.

I evaluate the various MARP value with diverse Hadoop benchmarks. The experiments reports different optimal MARP value for diverse sizes of the job and of the workloads, but exposes a similar non-monotone behavior—*i.e.*, the MARP value, either higher or lower than the optimal value, all will degrade Hadoop performance. Based on the analysis of memory consumption in Hadoop cluster, we find that the static value of MARP may limit the Hadoop performance by disturbing the memory utilization. This can be divided into 3 related problems and we handle all of them in our self-balancing proposition:

- **Loss of Jobs Parallelism** (LoJP) is the problem caused by low MARP value. When Hadoop configuration defines a low value to MARP, that means only a few jobs can be scheduled into Hadoop cluster and are processed in parallel. At the moment, if the running jobs are all small jobs—*i.e.*, each running job only has a few tasks to process—the YarnChilds cannot consume all the available memory in Hadoop cluster, resulting in idle memory. In this case, even though

Hadoop cluster has additional free memory, no more jobs can be processed in parallel due to the low parallel number of running jobs, which is controlled by MARP. Therefore, the Hadoop performance certainly be degraded.

To address this problem, we propose an algorithm, in Section 5.1.3.1, to adjust MARP according to the free memory in Hadoop cluster. When there are free memory appearing, the MARP value will be elevated to permit more jobs can get into the cluster to process in parallel, thus improving Hadoop performance. This adjustment stops once all the free memory has been consumed.

- **Loss of Job Throughput** (LoJT) is the opposite problem of LoJP. This problem is caused by high MARP value. When MARP is set a high value, the parallel number of running jobs will also become larger and more jobs can process in parallel. Therefore, all the memory in Hadoop cluster is consumed by the jobs, and free memory does not exist anymore. However, the Hadoop performance is not necessarily good. Due to the high MARP value, when there are enough idle jobs waiting for processing, most of memory in Hadoop clsuter will be occupied by AppMasters, which are the first containers launched for these jobs by ResourceManager. As explained in Section 3.1, AppMaster is only a private controller for its corresponding job, but does not concern any task processing, which is responsible by YarnChild. If most of memory is allocated to AppMasters, that means only a few memory can be used by YarnChild for processing the jobs. The Hadoop performance therefore is also degraded but with a high memory utilization.

  To cope with this problem, the MARP value should be tuned down to limit the number of AppMasters, thus pushing more memory to YarnChilds for job processing. When the memory utilization in Hadoop cluster is higher than a given threshold, we can believe that LoJT problem has probably emerged. In this case, we can begin to decrease the MARP value to try to handle this problem. Decreasing MARP value will not stop until the memory utilization become lower than the given threshold. This algorithm is introduced in Section 5.1.3.2.

Based on the analysis of the two problems, we can find that the optimal MARP value will differ from each other according to the various characteristics of work-

loads (*e.g.*, the size of jobs or of workloads). In this case, Hadoop users require an new approach to dynamically tune MARP at runtime according to the dynamics of workloads. However, due to the different principles of the two problems, their solutions are against each other, thus maybe leading to oscillation between the two solutions. To prevent the two solutions from mutual interference, we combine them in a double-threshold algorithm to coordinate their different adjustments.

- The last problem is named **Large Drops of Memory Utilization**. This problem is caused by the job self-managing mechanism which is introduced by YARN itself. At the beginning or the end of job processing, only AppMaster run in Hadoop cluster, which consumes just a few of memory. Although this period is short, it still cause temporary free memory in Hadoop cluster, resulting in low transitory memory utilization. Based on our experiments, we find this problem make the memory utilization full of fluctuations (cf. Fig. 5.7) which seriously affects the detection of the accurate state of Hadoop memory consumption. Therefore, to handle this problem, we introduce a Kalman filter to smooth the memory utilization.

Thanks to the analysis of these three problems, we propose a self-balancing algorithms in Section 5.1 to dynamically balance job parallelism and throughput in Hadoop cluster to optimize the system performance when processing concurrent workloads.

6.2.2 Rapid Deployment Prototype

Beyond the self-balancing algorithm, we also develop a new tool (named *hadoop-benchmark* in Section 5.2) to accelerate the deployment of a complete Hadoop cluster as well as a set of well-known Hadoop benchmarks. Hadoop performance highly depends on the infrastructure of Hadoop cluster. Researchers therefore have to deploy their *ad hoc* cluster to reproduce the other approaches to compare with their works. However, deploying and maintaining a complete Hadoop cluster is a time-consuming task and requires lots of experience to handle various issues of Hadoop running on diverse infrastructure. Hadoop researchers must spend lots of time and energy on irrelevant works, but not on studies. Meanwhile, for the same reason, the rapid deployment of Hadoop cluster also attracts attention

from elasticity researches when concerning timeliness (how to quickly provision or un-provision Hadoop compute nodes). In this case, we propose a prototype tool (*i.e.*, *hadoop-benchmark*) to quickly construct an *ad hoc* infrastructure and then to deploy a complete Hadoop cluster on this infrastructure.

hadoop-benchmark is developed on Docker container technologies. Docker is an open-source project for automating the deployment of applications as software containers. The primary idea of Docker is resource isolation rather than virtualization. In this case, compared to standard virtual machines, Docker containers avoid the overhead of starting and maintaining a complete operating system, but only simulates and supports the necessary dependencies required by the deployed applications. Therefore, it can be considered as a lightweight deployment method to provision applications on diverse infrastructure.

To package a complete Hadoop cluster and the benchmarks, we provide three Docker images: (1) *Base* image that installs Hadoop with vanilla configuration and contains all the dependencies required by each Hadoop component, (2) *Extension* images are extensions of *base* images with customized Hadoop configurations to guarantee that all the Hadoop components in various nodes (Docker containers in this case) can compose a complete cluster, (3) *Benchmark* images are based on *Extension* image, which consists of set of well-known Hadoop benchmarks. Thanks to these Docker images, the Docker containers launched with them have been well configured, either in Hadoop platform or in benchmarks. However, deploying a well-configured Hadoop platform in Docker containers is far from reproducing a complete Hadoop cluster, because the Docker containers situating in different machines cannot connect to each other. To address this problem, we introduce Docker Swarm in *hadoop-benchmark*. Docker Swarm can create an overlay network across diverse machines to achieve the connection between Docker containers in these machines. Finally, the prototype of the complete Hadoop cluster deployed by *hadoop-benchmark* is shown in Figure 5.18. It probably contains three types of Docker containers: (1) hadoop-consul serving the key-value store to support the overlay network of Docker Swarm, (2) hadoop-controller acting as the master node of Hadoop cluster, and (3) a set of hadoop-compute that represents the compute nodes of Hadoop cluster. Furthermore, users can also launch another extra

Docker containers with *Benchmark* images to evaluate the Hadoop cluster and the approaches deployed on it.

6.3 Benefits

This thesis mainly focus on the platform- (Hadoop) and infrastructure- (OpenStack) level resource optimization, and reports three main benefits.

1. The first one is named CloudGC, a new middleware service integrated within the services supported by the various components of OpenStack. CloudGC can periodically browse the OpenStack instance manager to detect the *idle* instances, either *explicit* or *implicit* and stores them into two queues with different priorities. When the OpenStack Cloud becomes overcrowded and the user requirements research the Cloud infrastructure limitation, CloudGC can recycle the identified *idle* VM instances to free the occupied resources for the extra requirements, thereby improving the QoS as well as the resource utilization of Cloud infrastructure. When CloudGC cannot recycle enough resources for the new requirements, Cloud providers can fall back on Cloud elasticity to obtain additional resources. This does not only avoid unnecessary Cloud expenditure, but also elevate the effect of Cloud elasticity. Therefore, CloudGC can be considered as an important complementation to the Cloud elasticity.

2. The second benefit is a self-balancing approach for Hadoop cluster to optimize its performance when processing time-varying concurrent workloads. When Hadoop cluster process concurrent workloads, especially time-varying workloads, the static configuration of Hadoop may limit its performance. In this case, to eliminate the worthless memory consumption and idleness as well as to optimize the system performance, the self-balancing approach will dynamically balance the job parallelism and throughput, according to the dynamics of workloads. The primary idea of this approach is to provide the optimal Hadoop performance based on the provisioned resources. From this point of view, it is similar to CloudGC. This self-balancing approach can avoid unnecessary Hadoop expenditure and be able to improve the ef-

fect of Hadoop elasticity. It is certainly a complementation for the elasticity researches also focusing on Hadoop cluster.

3. The last one is a rapid deployment tool to quickly constitute a prototype of complete Hadoop cluster using Docker Container Technologies. This created prototype consists of two layers: (1) an *ad hoc* infrastructure composed by one overlay network (Docker Swarm) and several Docker containers (2) a well-configured Hadoop cluster and other Hadoop benchmarks as well as an implementation of the self-balancing approach which are all installed in the Docker containers. This tool can ensure the reproducibility of a complete Hadoop cluster and obviously accelerates the construction of its prototype. Furthermore, thanks to the lightweight deployment method (*i.e.*, Docker), users can easily achieve the rapid deployment of Hadoop components by modifying the Docker images. This is important to the timeliness of Hadoop elasticity, and therefore, this research can also be regarded as a complementation to Hadoop elasticity researches.

The two first benefits can effort to improve the resource utilization, thereby saving the cost or making the system provide the optimal performance based on the provisioned resources. Meanwhile, they can also optimize the effect of elasticity. The last prototype tool does not only benefit the acceleration of Hadoop deployment, but it also be possible to be directly used by elasticity. In summary, all these benefits achieve the self-optimization of resources in platform- and infrastructure-level, and can be considered as the complements to elasticity researches.

Chapter 7

Perspectives

This thesis concerns the self-optimization of platform- and infrastructure-level resources in Cloud computing. This self-optimization aims at achieving automatic resource monitoring, analyzing, planning, and executing in multi-layers. Therefore, the optimization solution mentioned in this thesis is separated into two different researches in platform and infrastructure, respectively. The main purpose of these researches is to automatically optimize the resource utilization of the platform or infrastructure, thereby improving the system performance or resource utilization as well as avoiding unnecessary additional expenditure.

The approaches introduced in this thesis can be considered as early efforts at resource management for Big Data platform in Cloud computing. Although the evaluations have proven their initial efforts of resource optimization for platform performance and infrastructure utilization, the follow-up works should continue to improve the approaches, thereby adapting them to generalized and complex cases. The plans for the future works are presented in the following sections.

7.1 Short Term

Derived from this thesis, the short term perspectives to improve these researches in multi-layers will remain within the 2 primary research lines: **Infrastructure**- and **Platform**- level resource optimizations. The central idea for improving and developing the two researches is common: **Generality**—*i.e.*, these researches should be improved and developed to be widely adapted to various realistic situations, with high QoS. The generality for the resource optimizations in different layers are also diverse. For infrastructure-level, the principal works are to

improve the recycling service, thus makes CloudGC not only achieve the services but also operates with high reliability and QoS in complex usage environments. In Hadoop platform, the resource optimization with self-balancing algorithm focuses only on the memory consumption. The consequence is that it cannot figure out and solve the performance issues caused by other resources, such as CPU or I/O throughput. In this case, the future researches about Hadoop cluster should focus on the various type of resources, while integrating these diverse resource optimizations together. The details of these future works at short term will be further explained in the next two sections.

7.1.1 CloudGC for OpenStack

At the moment, CloudGC is only integrated in OpenStack and provides an initial recycling service to free the occupied resources in *idle* VM instances for supporting new user requirements, when the previous user requirements has reached Cloud infrastructure limitation. Benefiting from this CloudGC, the user requirements submitted to an OpenStack Cloud are able to exceed the infrastructure limitation, thereby making the resources able to serve other urgent requirements, improving the resource utilization in Cloud infrastructure. However, this is far from enough to be regarded as a complete middleware service which can be supplied to either Cloud providers or end-users. In this case, these future works at short term aim to improve CloudGC by achieving the additional services which will be introduced below, thereby adapting CloudGC to various Cloud computing, such as EC2.

- As introduced before, OpenStack is a complex project consisting of a set of components. Beyond managing the resources and VMs in Cloud computing, it also provides various services for their Cloud users, such as volume service (**Cinder**), network service (**Neutron**), and various object storage service (**Swift**), etc. This makes end-users able to customize their VM instances in diverse aspects like provisioning additional volume to VM disk, specific network setting of VM, etc. However, the current CloudGC only supports an initial recovery service supported by OpenStack—*i.e.*, the snapshots for recycled VM instances are just the images created by **Glance**, which only contain the in-

formation inside the VM instances, such as the operating system configuration, installed softwares, and all the data stored in initial disk, etc. The additional customizations (such as additional volume) will not be backed up and certainly are not able to be resumed, when CloudGC is required to recover the associated recycled VM, because these information has lost when the VM instance is removed from the Cloud. Therefore, based on the current implementation, CloudGC requires further developments to achieve comprehensive snapshot for supporting flawless recovery service. The new snapshot should be able to back up all the associated customizations of the VM instance which is identified to be recycled—*i.e.*, not only the information inside VM, but also the settings surrounding the instance.

- When CloudGC finds a new *idle* VM instance, it only snapshots the *explicit idle* VM instance but leaves the *implicit* ones to be backed up on-demand. This is due to the active state of *implicit idle* VM instance, which allows users to freely access to the VM for any activities or modifications. In this case, the snapshots of these VM instances are prone to become "ancient". The current snapshot service of OpenStack only supports creating a new image to once back up everything inside the VM instance. In this case, this will lead to 2 issues: (1) the new image will take lots of time and network bandwidth to back up the information, which have been snapshotted in "ancient" image, (2) the "ancient" image become useless and may occupy the volume in Cloud infrastructure, (3) the frequent backup process may disturb the network to affect the normal operation of Cloud and VM instances. Furthermore, as shown in Figure 4.9, even though the on-demand snapshots can avoid these negative impacts of CloudGC caused by *implicit idle* VM instances, it also obviously delays the recycling process and thus degrades the performance of CloudGC. Therefore, incremental snapshot service becomes an interesting solution in future works. It should be able to reuse the "ancient" snapshots to prevent the backup of snapshotted information. It cannot only solve the problem of reduntant backup of *implicit idle* VM instances, but also permits CloudGC to give up on-demand snapshots, thereby improving the performance and QoS of CloudGC.

- The last future work of CloudGC is also associated to *implicit idle* VM instances. To detect the *implicit idle* VM instances, CloudGC requires the CPU activities of the VM instances from **Ceilometer** and compares them with a given threshold. When the CPU activities of target VM instance is lower than the threshold for a long-enough period, it will be considered as *implicit idle*. However, the detection only depending on the CPU activities is not reliable to determine whether a VM instance is idle. For memory or I/O throughput intensive workloads, their CPU activities may keep in a low level. Furthermore, if these workloads can continue for a long-enough period while keeping low CPU activities, the VM instance is possible to be determined as *idle* one by CloudGC, thereby causing CloudGC to recycle non-idle VM instances and degrading the QoS of Cloud computing. Therefore, a new metrics for detecting *implicit idle* VM instances should be proposed to avoid these mistakes.

7.1.2 Self-balancing algorithm for Hadoop

The self-balancing algorithm proposed in Section 5.1 aims at adjusting Hadoop configuration (*i.e.*, MARP) according to the dynamics of workloads, to optimize the memory utilization, thereby improving Hadoop performance when processing concurrent workloads. This approach succeeds to treat the potential memory leaks caused by stiff memory allocation for AppMasters. However, the memory is not the only available and controllable resource affecting system performance in Hadoop cluster. In this case, the future works about Hadoop cluster should focus on the consumption optimization of other resources, and then joins these new researches to the self-balancing algorithm.

- Besides the memory, the amount of vCPU cores in Hadoop cluster, which is assigned for processing workloads, also affects the system performance. Since Hadoop 2.2, YARN has proposed 3 new parameters to control the amount of vCPUs for each map task, reduce task or AppMaster. At the moment, while processing MapReduce jobs, the number of vCPU cores for MapReduce tasks also become controllable in Hadoop cluster. The management of Hadoop cluster is thus not strictly limited to the memory, but may depend on another choice—*i.e.*, the approach for improving Hadoop performance is able to simultaneously

adjust the memory and CPU utilization to achieve a better performance improvement than memory-only optimization for MapReduce paradigm.

However, the defect for the CPU optimization can also be predicted. The three new parameters are only available to MapReduce paradigm. Therefore, this makes the future work cannot be widely adopted by YARN-based applications, except MapReduce framework.

- The I/O throughput of HDFS is another element which can significantly affect the Hadoop performance. For I/O intensive jobs, the majority of job completion time is occupied by accessing the data in disk—*i.e.*, when a task has been scheduled into Hadoop cluster for data processing, it must firstly wait for the target data to be uploaded from disk to memory. In this case, how to shorten the upload time (*i.e.*, the time to take the data from disk to memory) has become an issue for performance optimization of Hadoop cluster.

  In the current version of Hadoop, many scheduling strategies are proposed to developers to shorten the upload time by reducing the data transmission among compute nodes in distributed system (*i.e.*, Hadoop cluster), such as data localization—*i.e.*, Hadoop prefers to transmit the tasks to the compute nodes for processing, where HDFS stores the target data, rather than transmits data. These solutions significantly shorten the data upload time by reducing the data transmission in network, because the time for transmission tasks is much shorter than that for data. However, it cannot solve another bottleneck of upload time caused by I/O throughput—*i.e.*, uploading large amount of data from disk to memory will also take a long time.

  However, when the I/O intensive jobs process a set of related data, if Hadoop can upload the follow-up data to memory while processing the current task, the subsequent tasks probably can directly begin the processing without any waiting. This may obviously reduce the job completion time, thus improving the system performance.

These researches at short term aims at enriching the current works described in this thesis, and achieve good resource optimizations in respective layers. I believe that, thanks to the cooperation between these future researches and the

proposed approaches, the whole system (both platform and infrastructure) can automatically eliminate the resource leaks to maximize the resource utilization for the optimal performance which can be provided by the provisioned resources. Meanwhile the beneficiaries can also become more general rather than specific ones.

7.2 Long Term

In addition, I also propose 2 long term perspectives for the self-optimizations of platform- and infrastructure-level resources in Cloud computing. From the short term objectives proposed in above section, we can find that the researches still focus on the improvement of resource optimization in various layers, respectively. The combination of the self-optimization approaches in two layers (platform and infrastructure) should be an important issue for the performance optimization of Big Data platform in Cloud computing. Furthermore, even though the proposed approaches and the short term researches can provide the optimal performance for the users, they cannot ensure that the performance can fit the user requirements. In the case that all efforts have been done (*i.e.*, no resource leaks emerge in system, the configuration has been optimized according to the dynamics of workloads etc), provisioning new resources should be considered and be introduced to guarantee or improve the system performance for satisfying the QoS required by users, such as elasticity.

1. At the moment, the resource optimizations on platform and infrastructure are independent. This may lead to conflicts between the two optimization approaches. That means, if the two optimization approaches are applied together in one system, it is difficult to determine whether they can coordinate each other to improve performance. When CloudGC recycles several *idle* VM instances which are used as compute nodes in a Hadoop cluster, this may make the self-balancing approach get wrong memory records from ResourceManager—*i.e.*, the cluster information in ResourceManager is not instantaneously updated if these compute nodes are not removed from Hadoop cluster by ResourceManager, thereby resulting in erroneous adjustments to

cause negative impact in Hadoop cluster. In the cooperation of the two approaches in one system, there might be lots of such issues which require frequent coordination and specific developments. Moreover, the different optimization approaches also have their own resource monitoring and adjusting components. This may cause redundant work and thus degrade the respective robustness of the approaches, even resulting in negative impact on the target system. In this case, the integration of the resource optimizations in different layers becomes an important issue for the Big Data platform in Cloud computing. In particular, avoiding conflicts and unified management of resources in different layers are 2 core concerns in this future research.

2. The proposed approaches of resource optimizations can only ensure the optimal performance or maximize the resource utilization, based on the provisioned resources. Although these approaches succeed to guarantee and even improve the system performance without additional expenditure, it is possible that they are still not able to ensure that the optimal performance (based on provisioned resources) can satisfy the *Service-Level Agreement* (SLA) or QoS required by users. Therefore, to ensure that the system performance can truly satisfy user requirements, enough new resources should be provisioned into the system to guarantee and improve the performance, especially in the case that all the resource leaks have been eliminated. In this case, when the new resources are necessary and how much resources are needed are 2 issues for the system, which are also the core problems of elasticity researches. As I mentioned before, the proposed approaches can be considered as complements to elasticity researches both in platform- and infrastructure-level. Next to the resource optimization of provisioned resources, how to fall back on the elasticity to guarantee the performance in a reasonable level should be therefore a core concern in Cloud computing. This is obvious that, beyond the integration of diverse resource optimizations in various layers, the integration between these self-optimization approaches and elasticity researches are also an important issue affecting the Hadoop performance in Cloud computing.

From my point of view, besides the improvements of two approaches (*i.e.*, self-balancing approach and CloudGC) of resource optimization in different layers, the two long term perspectives do not only need the profound understanding on both Hadoop platform and OpenStack, but also requires lots of experience on elasticity researches.

Bibliography

[1] Al-Shishtawy, A., and Vlassov, V. Elastman: Elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference* (New York, NY, USA, 2013), CAC '13, ACM, pp. 7:1–7:10.

[2] Alharkan, T., and Martin, P. Idsaas: Intrusion detection system as a service in public clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on* (May 2012), pp. 686–687.

[3] Amazon WS. Website `https://aws.amazon.com`.

[4] Appel, A. W. Simple generational garbage collection and fast allocation. *Software: Practice and Experience 19*, 2 (1989), 171–183.

[5] Aydin, H., Melhem, R., Mosse, D., and Mejia-Alvarez, P. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers 53*, 5 (May 2004), 584–600.

[6] Beernaert, L., Matos, M., Vilaça, R., and Oliveira, R. Automatic Elasticity in OpenStack. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management* (New York, NY, USA, 2012), SDMCMM '12, ACM, pp. 2:1–2:6.

[7] Beloglazov, A., and Buyya, R. OpenStack neat: A framework for dynamic consolidation of virtual machines in OpenStack clouds–A blueprint. *Cloud Computing and Distributed Systems (CLOUDS) Laboratory* (2012).

[8] Berekmeri, M., Serrano, D., Bouchenak, S., Marchand, N., and Robu, B. A control approach for performance of big data systems. In *IFAC World Congress* (2014).

[9] Bouchenak, S. Automated control for sla-aware elastic clouds. In *Proceedings of the Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks* (New York, NY, USA, 2010), FeBiD '10, ACM, pp. 27–28.

[10] Brebner, P. C. Is your cloud elastic enough?: Performance modelling the elasticity of infrastructure as a service (iaas) cloud applications. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2012), ICPE '12, ACM, pp. 263–266.

[11] Bryant, R., Tumanov, A., Irzak, O., Scannell, A., Joshi, K., Hiltunen, M., Lagar-Cavilla, A., and de Lara, E. Kaleidoscope: Cloud Micro-elasticity via VM State Coloring. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 273–286.

[12] Buyya, R., Beloglazov, A., and Abawajy, J. Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges. *arXiv preprint arXiv:1006.0308* (2010).

[13] Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A. M., and Doyle, R. P. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev. 35*, 5 (Oct. 2001), 103–116.

[14] Chen, K., Powers, J., Guo, S., and Tian, F. CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds. *IEEE Trans. Parallel Distrib. Syst.* (2014).

[15] Chen, Y., Ganapathi, A., Griffith, R., and Katz, R. H. The Case for Evaluating MapReduce Performance Using Workload Suites. In *IEEE/ACM MASCOTS* (2011).

[16] Chiu, D., Shetty, A., and Agrawal, G. Elastic cloud caches for accelerating service-oriented computations. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[17] Corradi, A., Fanelli, M., and Foschini, L. VM consolidation: A real case based on OpenStack Cloud. *Future Generation Computer Systems 32* (2014), 118–127.

[18] Das, S., Agrawal, D., and El Abbadi, A. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst. 38*, 1 (Apr. 2013), 5:1–5:45.

[19] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. Twister: A Runtime for Iterative MapReduce. In *HPDC* (2010).

[20] Farokhi, S., Lakew, E. B., Klein, C., Brandic, I., and Elmroth, E. Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints. In *2015 International Conference on Cloud and Autonomic Computing (ICCAC)* (Sept 2015), pp. 69–80.

[21] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. *IBM technology 28* (2014), 32.

[22] Ghit, B., Yigitbasi, N., Iosup, A., and Epema, D. H. J. Balanced resource allocations across multiple dynamic MapReduce clusters. In *ACM SIGMETRICS* (2014).

[23] Guo, Y., Rao, J., and Zhou, X. ishuffle: Improving hadoop performance with shuffle-on-write. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (2013).

[24] He, S., Guo, L., Ghanem, M., and Guo, Y. Improving resource utilisation in the cloud environment using multivariate probabilistic models. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (2012), IEEE, pp. 574–581.

[25] Heintz, B., Chandra, A., Sitaraman, R., and Weissman, J. End-to-end optimization for geo-distributed mapreduce. *Cloud Computing, IEEE Transactions on PP*, 99 (2014), 1–1.

[26] Herbst, N. R., Kounev, S., and Reussner, R. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (San Jose, CA, 2013), USENIX, pp. 23–27.

[27] Herodotou, H., and Babu, S. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB 4*, 11 (2011).

[28] Herodotou, H., Lim, H., Luo, G., and Borisov, N. Starfish: A Self-tuning System for Big Data Analytics. *Conference on Innovative Data Systems Research* (2011).

[29] Hong, H. J., Chen, D. Y., Huang, C. Y., Chen, K. T., and Hsu, C. H. Placing Virtual Machines to Optimize Cloud Gaming Experience. *IEEE Transactions on Cloud Computing 3*, 1 (Jan 2015), 42–53.

[30] Hong, S., Ravindra, P., and Anyanwu, K. Adaptive information passing for early state pruning in mapreduce data processing workflows. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (2013).

[31] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)* (Mar. 2010), IEEE, pp. 41–51.

[32] IBM. An Architectural Blueprint for Autonomic Computing, 4. edition. Tech. rep., IBM, 2006.

[33] Jahani, E., Cafarella, M. J., and Ré, C. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.* (2011).

[34] Jones, R., and Lins, R. D. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[35] Kim, I. K., Zeng, S., Young, C., Hwang, J., and Humphrey, M. A Supervised Learning Model for Identifying Inactive VMs in Private Cloud Data Centers. In *The 2016 ACM/IFIP/USENIX International Middleware Conference, Industry Track (Middleware'16)* (Trento, Italy, December 2016).

[36] Lama, P., and Zhou, X. AROMA: automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC* (2012).

[37] Li, C., Zhuang, H., Lu, K., Sun, M., Zhou, J., Dai, D., and Zhou, X. An Adaptive Auto-configuration Tool for Hadoop. In *ICECCS* (2014).

[38] Liao, G., Datta, K., and Willke, T. L. Gunther: Search-Based Auto-tuning of MapReduce. In *Euro-Par 2013 Parallel Processing* (2013).

[39] Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM 26*, 6 (June 1983), 419–429.

[40] Liu, J., Ravi, N., Chakradhar, S., and Kandemir, M. Panacea: Towards Holistic Optimization of MapReduce Applications. In *CGO* (2012).

[41] Meisner, D., Gold, B. T., and Wenisch, T. F. Powernap: eliminating server idle power. *Acm Sigplan Notices 44*, 3 (2009), 205–216.

[42] Mell, P., and Grance, T. *The NIST definition of cloud computing.* Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

[43] Moreno-Vozmediano, R., Montero, R. S., and Llorente, I. M. Elastic management of cluster-based services in the cloud. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds* (New York, NY, USA, 2009), ACDC '09, ACM, pp. 19–24.

[44] Niu, S., Zhai, J., Ma, X., Tang, X., and Chen, W. Cost-effective cloud hpc resource provisioning by building semi-elastic virtual clusters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 56:1–56:12.

[45] Nzekwa, R., Rouvoy, R., and Seinturier, L. A flexible context stabilization approach for self-adaptive application. In *Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2010), IEEE.

[46] OpenSatck. Open source software for creating private and public clouds. Website `https://www.openstack.org/`, 2016.

[47] Padala, P., Hou, K., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A. Automated control of multiple virtualized resources. In *Proceedings of the 2009 EuroSys* (2009).

[48] Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2007 EuroSys* (2007).

[49] Pandey, S., Wu, L., Guru, S. M., and Buyya, R. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *Advanced information networking and applications (AINA), 2010 24th IEEE international conference on* (2010), IEEE, pp. 400–407.

[50] Polo, J., Becerra, Y., Carrera, D., Torres, J., Ayguade, E., and Steinder, M. Adaptive mapreduce scheduling in shared environments. In *Cluster, Cloud and Grid Computing, 14th IEEE/ACM International Symposium on* (2014), pp. 61–70.

[51] Quan, D. M., Basmadjian, R., De Meer, H., Lent, R., Mahmoodi, T., Sannelli, D., Mezza, F., Telesca, L., and Dupont, C. Energy efficient resource allocation strategy for cloud data centres. In *Computer and information sciences II.* Springer, 2011, pp. 133–141.

[52] Ren, K., Gibson, G., Kwon, Y., Balazinska, M., and Howe, B. Hadoop's Adolescence; A Comparative Workloads Analysis from Three Research Clusters. In *SC Companion: High Performance Computing, Networking Storage and Analysis* (2012).

[53] Romano, P. Elastic, scalable and self-tuning data replication in the cloud-tm platform. In *Proceedings of the 1st European Workshop on Dependable Cloud Computing* (New York, NY, USA, 2012), EWDCC '12, ACM, pp. 5:1–5:2.

[54] Shen, Z., Subbiah, S., Gu, X., and Wilkes, J. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 5:1–5:14.

[55] Spinner, S., Kounev, S., Zhu, X., Lu, L., Uysal, M., Holler, A., and Griffith, R. Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems* (Sept 2014), pp. 157–166.

[56] Turowski, M., and Lenk, A. Vertical Scaling Capability of OpenStack. In *Service-Oriented Computing-ICSOC 2014 Workshops* (2015), Springer, pp. 351–362.

[57] Ungar, D. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Sigplan Notices* (1984), vol. 19, ACM, pp. 157–167.

[58] Wake-on-LAN. Website `https://en.wikipedia.org/wiki/Wake-on-LAN`.

[59] Wang, Y., Wang, X., Chen, M., and Zhu, X. Power-Efficient Response Time Guarantees for Virtualized Enterprise Servers. In *Real-Time Systems Symposium* (2008).

[60] Xu, L., Liu, J., and Wei, J. Fmem: A fine-grained memory estimator for mapreduce jobs. In *Proceedings of the 10th International Conference on Autonomic Computing* (2013).

[61] Yu, L., and Thain, D. Resource management for elastic cloud workflows. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on* (May 2012), pp. 775–780.

[62] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. Ecosystem: Managing energy as a first class operating system resource. *SIGARCH Comput. Archit. News 30*, 5 (Oct. 2002), 123–132.

[63] Zhang, B., Krikava, F., Rouvoy, R., and Seinturier, L. Self-configuration of the number of concurrently running mapreduce jobs in a hadoop cluster. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on* (July 2015), pp. 149–150.

[64] Zhang, B., Křikava, F., Rouvoy, R., and Seinturier, L. Self-Balancing Job Parallelism and Throughput in Hadoop. In *16th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)* (Heraklion, Crete, Greece, June 2016), E. Kalyvianaki and M. Jelasity, Eds., vol. 9687 of *Proceedings of the 16th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, Springer, pp. 129–143.

[65] Zhang, W., Rajasekaran, S., Wood, T., and Zhu, M. MIMP: Deadline and Interference Aware Scheduling of Hadoop Virtual Machines. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2014).

[66] Zhang, X., Kunjithapatham, A., Jeong, S., and Gibbs, S. Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mob. Netw. Appl. 16*, 3 (June 2011), 270–284.

[67] Zhi, J., Bila, N., and de Lara, E. Oasis: energy proportionality with hybrid server consolidation. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), pp. 10:1–10:13.

Appendix A

The Absolute Completion Time of Hadoop Benchmarks

	Wordcount-30MB	Terasort-30MB	Sort-30MB
10% (default)	1343.082s	178.125s	236.740s
Best-Static MARP Value	1020.742s	170.112s	195.311s
Dynamic MARP Value	980.39s	147.843s	183.610s

Table A.1: Mean Job Completion Time of 100 jobs in HiBench (s) - 1.

	Wordcount-3GB	Terasort-3GB	Sort-3GB
10% (default)	2941.693s	1845.816s	2719.682s
Best-Static MARP Value	2538.681s	1607.705s	2665.288s
Dynamic MARP Value	2435.7218s	1478.498s	2461.311s

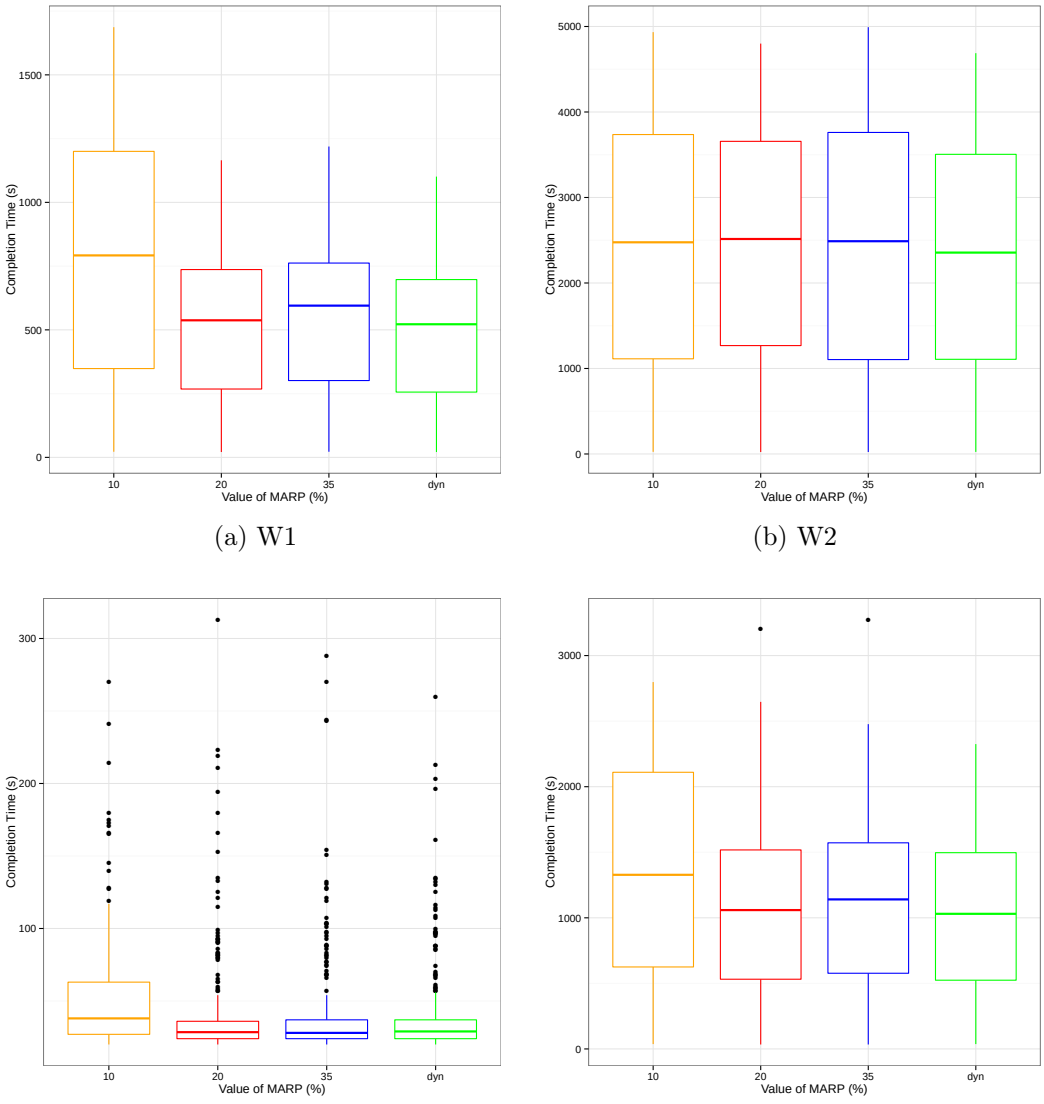
Table A.2: Mean Job Completion Time of 100 jobs in HiBench (s) - 2.

	10 Jobs	50 Jobs	100 Jobs	150 Jobs
10% (default)	473.259s	1159.537s	1798.396s	2697.594s
Best-Static MARP Value	444.863s	982.128s	1712.072s	2697.594s
Dynamic MARP Value	463.320s	962.416s	1561.007s	2594.27s

Table A.3: Mean Job Completion Time of Terasort-3GB under different stress conditions (s).

	10%	15%	20%	25%	30%	35%	40%	dyn
W1	3593s	3241s	3019s	3054s	3061s	3052s	3095s	2944s
W2	6991s	6733s	6650s	6732s	6783s	7024s	7055s	6286s
W3	4056s	3933s	3708s	3704s	3701s	3687s	3753s	3650s
W4	4726s	4379s	4202s	4249s	4244s	4300s	4497s	4053s

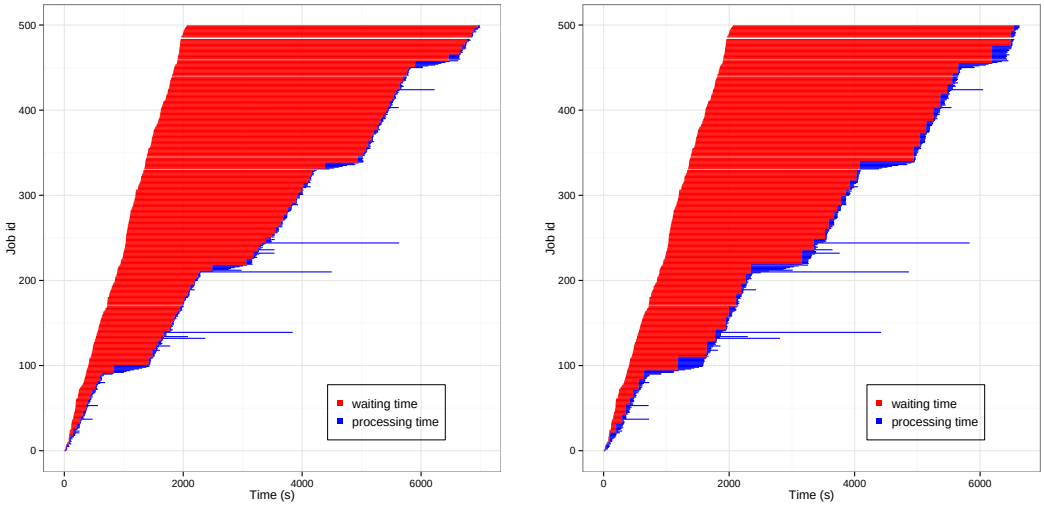
Table A.4: The Completion Time of SWIM Workloads (s).



(a) W1 (b) W2

(c) W3 (d) W4

Figure A.1: The distribution of Job Completion Time from 4 SWIM Workloads.



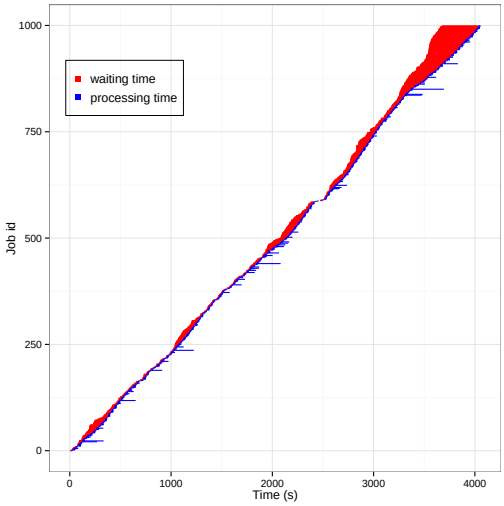
(a) MARP (10% default)

(b) MARP (20%)

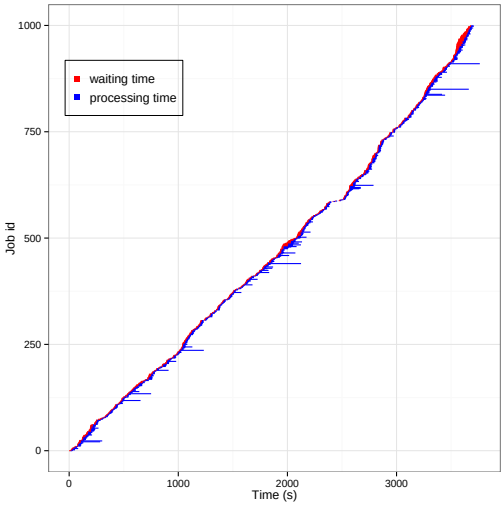
(c) MARP (35%)

(d) MARP (dynamic)

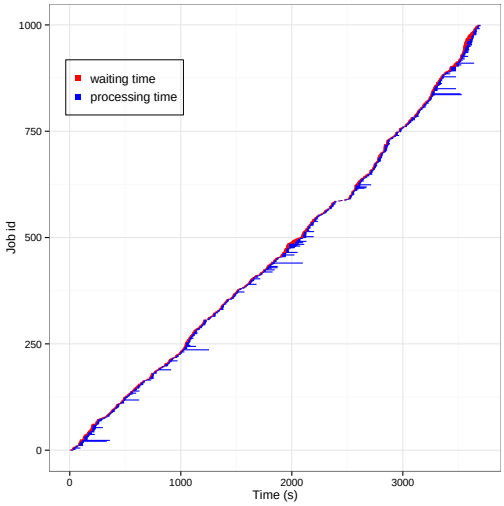
Figure A.2: The details of Job Completion Time in SWIM Workloads 2.



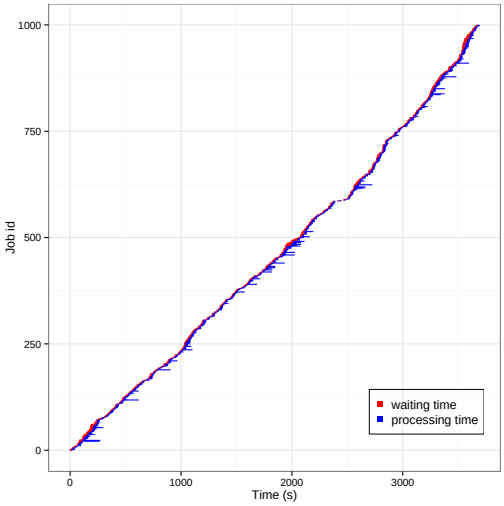
(a) MARP (10% default)



(b) MARP (20%)

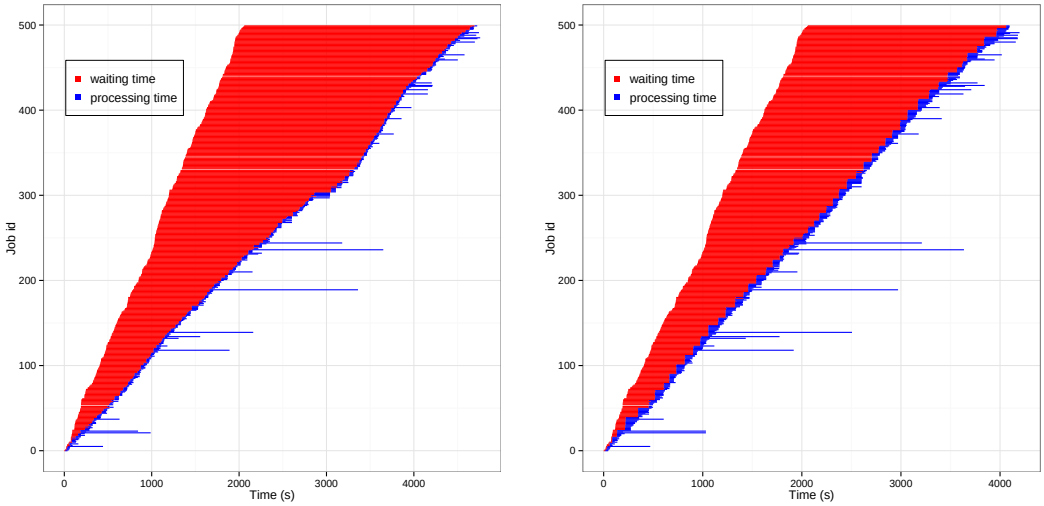


(c) MARP (35%)



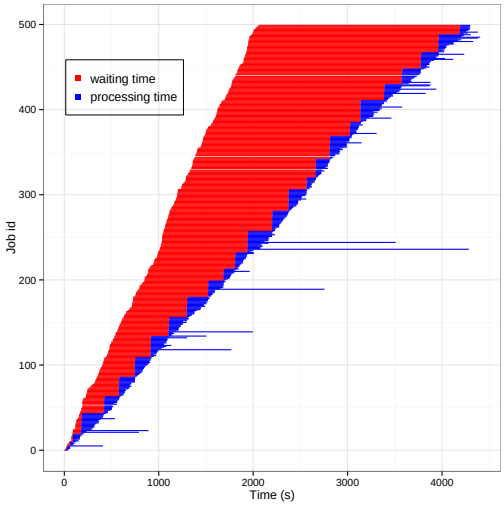
(d) MARP (dynamic)

Figure A.3: The details of Job Completion Time in SWIM Workloads 3.

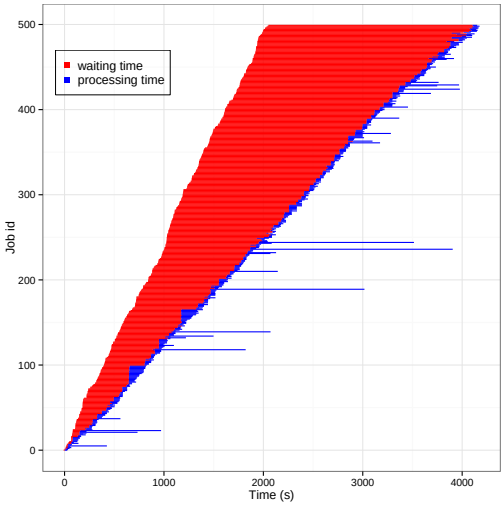


(a) MARP (10% default)

(b) MARP (20%)

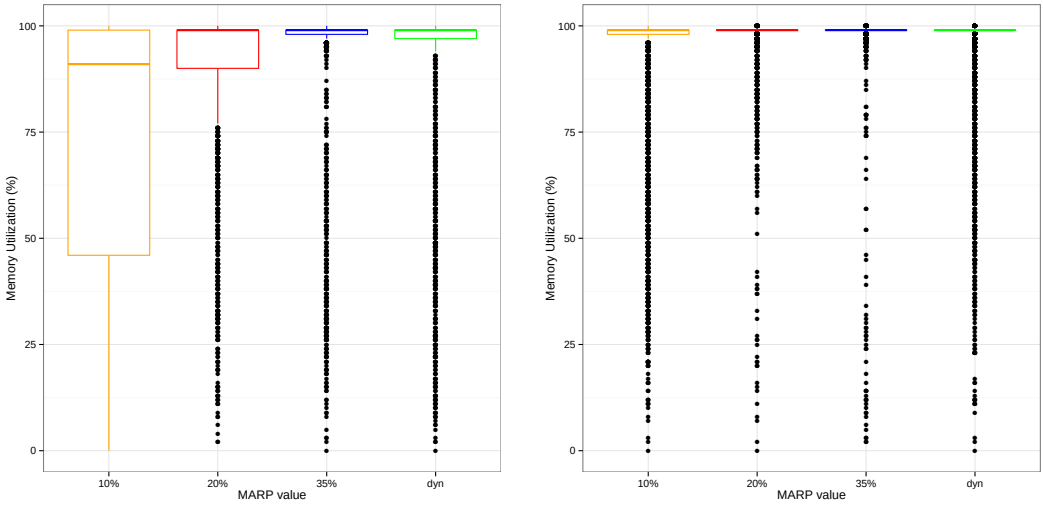


(c) MARP (35%)



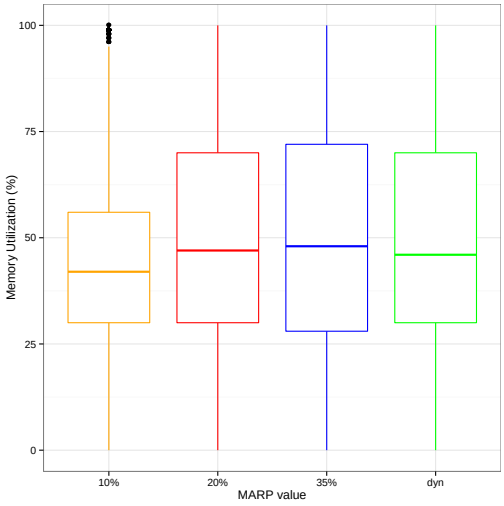
(d) MARP (dynamic)

Figure A.4: The details of Job Completion Time in SWIM Workloads 4.

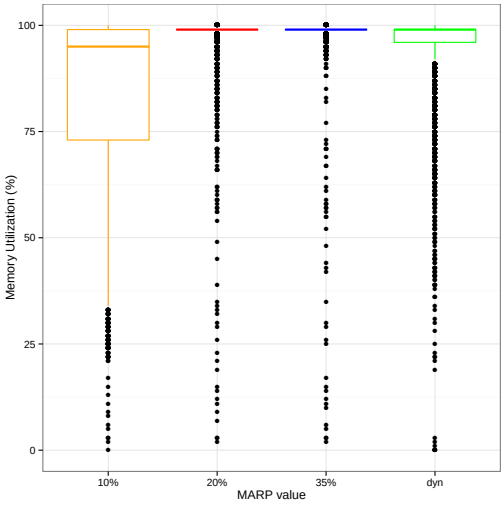


(a) W1

(b) W2



(c) W3



(d) W4

Figure A.5: The Memory Consumption of SWIM Workloads in each second.